Table of Contents

Getting started

Introduction	1.1
Contents of the book	1.2
"Why use F#?" in one page	1.3
Installing and using F#	1.4
F# syntax in 60 seconds	1.5
Learning F#	1.6
Troubleshooting F#	1.7
Low-risk ways to use F# at work	1.8
Twenty six low-risk ways to use F# at work	1.8.1
Using F# for development and devops scripts	1.8.2
Using F# for testing	1.8.3
Using F# for database related tasks	1.8.4
Other interesting ways of using F# at work	1.8.5

Why use F#?

The "Why use F#?" Series	2.1
Introduction to the 'Why use F#' series	2.1.1
F# syntax in 60 seconds	2.1.2
Comparing F# with C#: A simple sum	2.1.3
Comparing F# with C#: Sorting	2.1.4
Comparing F# with C#: Downloading a web page	2.1.5
Four Key Concepts	2.1.6
Conciseness	2.1.7
Type inference	2.1.8
Low overhead type definitions	2.1.9
Using functions to extract boilerplate code	2.1.10
Using functions as building blocks	2.1.11

Pattern matching for conciseness 2.1.12
Convenience 2.1.13
Out-of-the-box behavior for types 2.1.14
Functions as interfaces 2.1.15
Partial Application 2.1.16
Active patterns 2.1.17
Correctness 2.1.18
Immutability 2.1.19
Exhaustive pattern matching 2.1.20
Using the type system to ensure correct code 2.1.21
Worked example: Designing for correctness 2.1.22
Concurrency 2.1.23
Asynchronous programming 2.1.24
Messages and Agents 2.1.25
Functional Reactive Programming 2.1.26
Completeness 2.1.27
Seamless interoperation with .NET libraries 2.1.28
Anything C# can do... 2.1.29
Why use F#: Conclusion 2.1.30

Thinking Functionally

The "Thinking Functionally" Series 3.1
Thinking Functionally: Introduction 3.1.1
Mathematical functions 3.1.2
Function Values and Simple Values 3.1.3
How types work with functions 3.1.4
Currying 3.1.5
Partial application 3.1.6
Function associativity and composition 3.1.7
Defining functions 3.1.8
Function signatures 3.1.9
Organizing functions 3.1.10
Attaching functions to types 3.1.11

Worked example: A stack based calculator 3.1.12

Understanding F#

The "Expressions and syntax" Series 4.1
Expressions and syntax: Introduction 4.1.1
Expressions vs. statements 4.1.2
Overview of F# expressions 4.1.3
Binding with let, use, and do 4.1.4
F# syntax: indentation and verbosity 4.1.5
Parameter and value naming conventions 4.1.6
Control flow expressions 4.1.7
Exceptions 4.1.8
Match expressions 4.1.9
Formatted text using printf 4.1.10
Worked example: Parsing command line arguments 4.1.11
Worked example: Roman numerals 4.1.12
The "Understanding F# types" Series 4.2
Understanding F# types: Introduction 4.2.1
Overview of types in F# 4.2.2
Type abbreviations 4.2.3
Tuples 4.2.4
Records 4.2.5
Discriminated Unions 4.2.6
The Option type 4.2.7
Enum types 4.2.8
Built-in .NET types 4.2.9
Units of measure 4.2.10
Understanding type inference 4.2.11
Choosing between collection functions 4.3
The "Object-oriented programming in F#" Series 4.4
Object-oriented programming in F#: Introduction 4.4.1
Classes 4.4.2
Inheritance and abstract classes 4.4.3

Interfaces 4.4.4
Object expressions 4.4.5
The "Computation Expressions" Series 4.5
Computation expressions: Introduction 4.5.1
Understanding continuations 4.5.2
Introducing 'bind' 4.5.3
Computation expressions and wrapper types 4.5.4
More on wrapper types 4.5.5
Implementing a builder: Zero and Yield 4.5.6
Implementing a builder: Combine 4.5.7
Implementing a builder: Delay and Run 4.5.8
Implementing a builder: Overloading 4.5.9
Implementing a builder: Adding laziness 4.5.10
Implementing a builder: The rest of the standard methods 4.5.11
Organizing modules in a project 4.6
The "Dependency cycles" Series 4.7
Cyclic dependencies are evil 4.7.1
Refactoring to remove cyclic dependencies 4.7.2
Cycles and modularity in the wild 4.7.3
The "Porting from C#" Series 4.8
Porting from C# to F#: Introduction 4.8.1
Getting started with direct porting 4.8.2

Functional Design

The "Designing with types" Series 5.1
Designing with types: Introduction 5.1.1
Single case union types 5.1.2
Making illegal states unrepresentable 5.1.3
Discovering new concepts 5.1.4
Making state explicit 5.1.5
Constrained strings 5.1.6
Non-string types 5.1.7
Designing with types: Conclusion 5.1.8

Algebraic type sizes and domain modelling 5.2
Thirteen ways of looking at a turtle 5.3
Thirteen ways of looking at a turtle (part 2) 5.3.1
Thirteen ways of looking at a turtle - addendum 5.3.2

Functional Patterns

How to design and code a complete program 6.1
A functional approach to error handling (Railway oriented programming) 6.2
Railway oriented programming: Carbonated edition 6.2.1
The "Understanding monoids" Series 6.3
Monoids without tears 6.3.1
Monoids in practice 6.3.2
Working with non-monoids 6.3.3
The "Understanding Parser Combinators" Series 6.4
Understanding Parser Combinators 6.4.1
Building a useful set of parser combinators 6.4.2
Improving the parser library 6.4.3
Writing a JSON parser from scratch 6.4.4
The "Handling State" Series 6.5
Dr Frankenfunctor and the Monadster 6.5.1
Completing the body of the Monadster 6.5.2
Refactoring the Monadster 6.5.3
The "Map and Bind and Apply, Oh my!" Series 6.6
Understanding map and apply 6.6.1
Understanding bind 6.6.2
Using the core functions in practice 6.6.3
Understanding traverse and sequence 6.6.4
Using map, apply, bind and sequence in practice 6.6.5
Reinventing the Reader monad 6.6.6
Map and Bind and Apply, a summary 6.6.7
The "Recursive types and folds" Series 6.7
Introduction to recursive types 6.7.1
Catamorphism examples 6.7.2

Introducing Folds 6.7.3
Understanding Folds 6.7.4
Generic recursive types 6.7.5
Trees in the real world 6.7.6
The "A functional approach to authorization" Series 6.8
A functional approach to authorization 6.8.1
Constraining capabilities based on identity and role 6.8.2
Using types as access tokens 6.8.3

Testing

An introduction to property-based testing 7.1
Choosing properties for property-based testing 7.2

Examples and Walkthroughs

Worked example: Designing for correctness 8.1
Worked example: A stack based calculator 8.2
Worked example: Parsing command line arguments 8.3
Worked example: Roman numerals 8.4
Commentary on 'Roman Numerals Kata with Commentary' 8.5
Calculator Walkthrough: Part 1 8.6
Calculator Walkthrough: Part 2 8.6.1
Calculator Walkthrough: Part 3 8.6.2
Calculator Walkthrough: Part 4 8.6.3
Enterprise Tic-Tac-Toe 8.7
Enterprise Tic-Tac-Toe, part 2 8.7.1
Writing a JSON parser from scratch 8.8

Other

Ten reasons not to use a statically typed functional programming language 9.1
Why I won't be writing a monad tutorial 9.2
Is your programming language unreasonable? 9.3

We don't need no stinking UML diagrams 9.4
Introvert and extrovert programming languages 9.5
Swapping type-safety for high performance using compiler directives 9.6

F# for Fun and Profit eBook

This is an eBook version of my site fsharpforfunandprofit.com, created by popular request for people who want to read it offline on their kindle or phone. Special thanks to Roman Provaznik for the final nag.

The site (and this book) aims to introduce you to F# and show you ways that F# can help in day-to-day development of mainstream commercial business software. On the way, I hope to open your mind to the joys of functional programming - it really is fun!

Many of the posts were not really designed to be in a book, but I have arranged them so that beginner posts come first, and more advanced posts later.

About F#

If you are completely new to F#, F# is a general purpose functional/hybrid programming language which is great for tackling almost any kind of software challenge.

F# is free and open source, and runs on Linux, Mac, Windows and more. To download and install F#, or to find out more, go to F# Foundation site at fsharp.org.

Getting started

Next, before randomly dipping into the posts, you should read the "why use F#?" page and then the whole "why use F#" series. After that the "site contents" page provides suggestions for further reading on functions, types and more.

There is a page with some advice on learning F#, and if you have problems trying to get your code to compile, the troubleshooting F# page might be helpful.

I will assume that you do not need instruction in the basics of programming and that you are familiar with C#, Java, or a similar C-like language. It will also be helpful if you are familiar with the Mono/.NET library.

On the other hand, I will not assume that you have a mathematical or computer science background. There will be no mathematical notation, and no mysterious concepts like "functor", "category theory" and "anamorphism". If you are already familiar with Haskell or ML, this is probably not the place for you!

Also, I will not attempt to cover highly technical or mathematical applications. F# is an excellent tool for these domains, but it requires an approach that is different from business software.

Getting started

- Installing and using F# will get you started.
- Why use F#? An interactive tour of F#.
- Learning F# has tips to help you learn more effectively.
- Troubleshooting F# for when you have problems getting your code to compile.

and then you can try...

- Twenty six low-risk ways to use F# at work. You can start right now -- no permission needed!

Tutorials

The following series are tutorials on the key concepts of F#.

- Thinking functionally starts from basics and explains how and why functions work the way they do.
- Expressions and syntax covers the common expressions such as pattern matching, and has a post on indentation.
- Understanding F# types explains how to define and use the various types, including tuples, records, unions, and options.
- Designing with types explains how to use types as part of the design process, making illegal states unrepresentable.
- Choosing between collection functions. If you are coming to F# from C#, the large number of list functions can be overwhelming, so I have written this post to help guide you to the one you want.
- Property-based testing: the lazy programmer's guide to writing 1000's of tests.
- Understanding computation expressions demystifies them and shows how you can create your own.

Functional patterns

These posts explain some core patterns in functional programming -- concepts such as "map", "bind", monads and more.

- Railway Oriented Programming: A functional approach to error handling
- State Monad: An introduction to handling state using the tale of Dr Frankenfunctor and the Monadster.

- Reader Monad: Reinventing the Reader monad.
- Map, bind, apply, lift, sequence and traverse: A series describing some of the core functions for dealing with generic data types.
- Monoids without tears: A mostly mathless discussion of a common functional pattern.
- Fold and recursive types: A look at recursive types, catamorphisms, tail recursion, the difference between left and right folds, and more.
- Understanding Parser Combinators: Creating a parser combinator library from scratch.
- Thirteen ways of looking at a turtle: demonstrates many different techniques for implementing a turtle graphics API, including state monads, agents, interpreters, and more!

Worked examples

These posts provide detailed worked examples with lots of code!

- Designing for correctness: How to make illegal states unrepresentable (a shopping cart example).
- Stack based calculator: Using a simple stack to demonstrate the power of combinators.
- Parsing commmand lines: Using pattern matching in conjunction with custom types.
- Roman numerals: Another pattern matching example.
- Calculator Walkthrough: The type-first approach to designing a Calculator.
- Enterprise Tic-Tac-Toe: A walkthrough of the design decisions in a purely functional implementation
- Writing a JSON Parser.

Specific topics in F#

General:

- Four key concepts that differentiate F# from a standard imperative language.
- Understanding F# indentation.
- The downsides of using methods.

Functions:

- Currying.
- Partial Application.

Control Flow:

- Match..with expressions and creating folds to hide the matching.
- If-then-else and loops.

- Exceptions.

Types:

- Option Types especially on why None is not the same as null.
- Record Types.
- Tuple Types.
- Discriminated Unions.
- Algebraic type sizes and domain modelling.

Controversial posts

- Is your programming language unreasonable? or, why predictability is important.
- Commentary on 'Roman Numerals Kata with Commentary'. My approach to the Roman Numerals Kata.
- Ten reasons not to use a statically typed functional programming language. A rant against something I don't get.
- We don't need no stinking UML diagrams or, why in many cases, using UML for class diagrams is not necessary.

Although F# is great for specialist areas such as scientific or data analysis, it is also an excellent choice for enterprise development. Here are five good reasons why you should consider using F# for your next project.

Conciseness

F# is not cluttered up with coding "noise" such as curly brackets, semicolons and so on.

You almost never have to specify the type of an object, thanks to a powerful type inference system.

And, compared with C#, it generally takes fewer lines of code to solve the same problem.

```
// one-liners
[1..100] |> List.sum |> printfn "sum=%d"

// no curly braces, semicolons or parentheses
let square x = x * x
let sq = square 42

// simple types in one line
type Person = {First:string; Last:string}

// complex types in a few lines
type Employee =
  | Worker of Person
  | Manager of Employee list

// type inference
let jdoe = {First="John";Last="Doe"}
let worker = Worker jdoe
```

Convenience

Many common programming tasks are much simpler in F#. This includes things like creating and using complex type definitions, doing list processing, comparison and equality, state machines, and much more.

And because functions are first class objects, it is very easy to create powerful and reusable code by creating functions that have other functions as parameters, or that combine existing functions to create new functionality.

```
// automatic equality and comparison
type Person = {First:string; Last:string}
let person1 = {First="john"; Last="Doe"}
let person2 = {First="john"; Last="Doe"}
printfn "Equal? %A"  (person1 = person2)

// easy IDisposable logic with "use" keyword
use reader = new StreamReader(..)

// easy composition of functions
let add2times3 = (+) 2 >> (*) 3
let result = add2times3 5
```

☑ Correctness

F# has a powerful type system which prevents many common errors such as null reference exceptions.

Values are immutable by default, which prevents a large class of errors.

In addition, you can often encode business logic using the type system itself in such a way that it is actually impossible to write incorrect code or mix up units of measure, greatly reducing the need for unit tests.

```
// strict type checking
printfn "print string %s" 123 //compile error

// all values immutable by default
person1.First <- "new name"  //assignment error

// never have to check for nulls
let makeNewString str =
   //str can always be appended to safely
   let newString = str + " new!"
   newString

// embed business logic into types
emptyShoppingCart.remove   // compile error!

// units of measure
let distance = 10<m> + 10<ft> // error!
```

Concurrency

F# has a number of built-in libraries to help when more than one thing at a time is happening. Asynchronous programming is very easy, as is parallelism.

F# also has a built-in actor model, and excellent support for event handling and functional reactive programming.

And of course, because data structures are immutable by default, sharing state and avoiding locks is much easier.

```
// easy async logic with "async" keyword
let! result = async {something}

// easy parallelism
Async.Parallel [ for i in 0..40 ->
      async { return fib(i) } ]

// message queues
MailboxProcessor.Start(fun inbox-> async{
    let! msg = inbox.Receive()
    printfn "message is: %s" msg
    })
```

⚒ Completeness

Although it is a functional language at heart, F# does support other styles which are not 100% pure, which makes it much easier to interact with the non-pure world of web sites, databases, other applications, and so on.

In particular, F# is designed as a hybrid functional/OO language, so it can do virtually everything that C# can do.

Of course, F# is part of the .NET ecosystem, which gives you seamless access to all the third party .NET libraries and tools. It runs on most platforms, including Linux and smart phones (via Mono and the new .NET Core).

Finally, it is well integrated with Visual Studio (Windows) and Xamarin (Mac), which means you get a great IDE with IntelliSense support, a debugger, and many plug-ins for unit tests, source control, and other development tasks. Or on Linux, you can use the MonoDevelop IDE instead.

```
// impure code when needed
let mutable counter = 0

// create C# compatible classes and interfaces
type IEnumerator<'a> =
    abstract member Current : 'a
    abstract MoveNext : unit -> bool

// extension methods
type System.Int32 with
    member this.IsEven = this % 2 = 0

let i=20
if i.IsEven then printfn "'%i' is even" i

// UI code
open System.Windows.Forms
let form = new Form(Width= 400, Height = 300,
   Visible = true, Text = "Hello World")
form.TopMost <- true
form.Click.Add (fun args-> printfn "clicked!")
form.Show()
```

The "Why Use F#?" series

The following series of posts demonstrates each of these F# benefits, using standalone snippets of F# code (and often with C# code for comparison).

- Introduction to the 'Why use F#' series. An overview of the benefits of F#
- F# syntax in 60 seconds. A very quick overview on how to read F# code
- Comparing F# with C#: A simple sum. In which we attempt to sum the squares from 1 to N without using a loop
- Comparing F# with C#: Sorting. In which we see that F# is more declarative than C#, and we are introduced to pattern matching.
- Comparing F# with C#: Downloading a web page. In which we see that F# excels at callbacks, and we are introduced to the 'use' keyword
- Four Key Concepts. The concepts that differentiate F# from a standard imperative language
- Conciseness. Why is conciseness important?
- Type inference. How to avoid getting distracted by complex type syntax
- Low overhead type definitions. No penalty for making new types
- Using functions to extract boilerplate code. The functional approach to the DRY principle
- Using functions as building blocks. Function composition and mini-languages make

code more readable

- Pattern matching for conciseness. Pattern matching can match and bind in a single step
- Convenience. Features that reduce programming drudgery and boilerplate code
- Out-of-the-box behavior for types. Immutability and built-in equality with no coding
- Functions as interfaces. OO design patterns can be trivial when functions are used
- Partial Application. How to fix some of a function's parameters
- Active patterns. Dynamic patterns for powerful matching
- Correctness. How to write 'compile time unit tests'
- Immutability. Making your code predictable
- Exhaustive pattern matching. A powerful technique to ensure correctness
- Using the type system to ensure correct code. In F# the type system is your friend, not your enemy
- Worked example: Designing for correctness. How to make illegal states unrepresentable
- Concurrency. The next major revolution in how we write software?
- Asynchronous programming. Encapsulating a background task with the Async class
- Messages and Agents. Making it easier to think about concurrency
- Functional Reactive Programming. Turning events into streams
- Completeness. F# is part of the whole .NET ecosystem
- Seamless interoperation with .NET libraries. Some convenient features for working with .NET libraries
- Anything C# can do.... A whirlwind tour of object-oriented code in F#
- Why use F#: Conclusion.

The F# compiler is a free and open source tool which is available for Windows, Mac and Linux (via Mono). Find out more about F# and how to install it at the F# Foundation.

You can use it with an IDE (Visual Studio, MonoDevelop), or with your favorite editor (VS Code and Atom have especially good F# support using Ionide), or simply as a standalone command line compiler.

If you don't want to install anything, you can try the .NET Fiddle site, which is an interactive environment where you can explore F# in your web browser. You should be able to run most of the code on this site there.

Working with the code examples

Once you have F# installed and running, you can follow along with the code samples.

The best way to run the code examples on this site is to type the code into an `.FSX` script file, which you can then send to the F# interactive window for evaluation. Alternatively you can type the examples directly into the F# interactive console window. I would recommend the script file approach for anything other than one or two lines.

For the longer examples, the code is downloadable from this website -- the links will be in the post.

Finally, I would encourage you to play with and modify the examples. If you then get compiler errors, do check out the "troubleshooting F#" page, which explains the most common problems, and how to fix them.

Projects and Solutions

F# uses exactly the same "projects" and "solutions" model that C# does, so if you are familiar with that, you should be able to create an F# executable quite easily.

To make a file that will be compiled as part of the project, rather than a script file, use the `.fs` extension. `.fsx` files will not be compiled.

An F# project does have some major differences from C# though:

- The F# files are organized *linearly*, not in a hierarchy of folders and subfolders. In fact, there is no "add new folder" option in an F# project! This is not generally a problem, because, unlike C#, an F# file contains more than one class. What might be a whole folder of classes in C# might easily be a single file in F#.
- The *order of the files in the project is very important*: a "later" F# file can use the public types defined in an "earlier" F# file, but not the other way around. Consequently, you

cannot have any circular dependencies between files.

- You can change the order of the files by right-clicking and doing "Move Up" or "Move Down". Similarly, when creating a new file, you can choose to "Add Above" or "Add Below" an existing file.

Shell scripts in F#

You can also use F# as a scripting language, rather than having to compile code into an EXE. This is done by using the FSI program, which is not only a console but can also be used to run scripts in the same way that you might use Python or Powershell.

This is very convenient when you want to quickly create some code without compiling it into a full blown application. The F# build automation system "FAKE" is an example of how useful this can be.

To see how you can do this yourself, here is a little example script that downloads a web page to a local file. First create an FSX script file -- call it "`ShellScriptExample.fsx`" -- and paste in the following code.

```
// ================================
// Description:
//    downloads the given url and stores it as a file with a timestamp
//
// Example command line:
//    fsi ShellScriptExample.fsx http://google.com google
// ================================

// "open" brings a .NET namespace into visibility
open System.Net
open System

// download the contents of a web page
let downloadUriToFile url targetfile =
    let req = WebRequest.Create(Uri(url))
    use resp = req.GetResponse()
    use stream = resp.GetResponseStream()
    use reader = new IO.StreamReader(stream)
    let timestamp = DateTime.UtcNow.ToString("yyyy-MM-ddTHH-mm")
    let path = sprintf "%s.%s.html" targetfile timestamp
    use writer = new IO.StreamWriter(path)
    writer.Write(reader.ReadToEnd())
    printfn "finished downloading %s to %s" url path

// Running from FSI, the script name is first, and other args after
match fsi.CommandLineArgs with
    | [| scriptName; url; targetfile |] ->
        printfn "running script: %s" scriptName
        downloadUriToFile url targetfile
    | _ ->
        printfn "USAGE: [url] [targetfile]"
```

Don't worry about how the code works right now. It's pretty crude anyway, and a better example would add error handling, and so on.

To run this script, open a command window in the same directory and type:

```
fsi ShellScriptExample.fsx http://google.com google_homepage
```

As you play with the code on this site, you might want to experiment with creating some scripts at the same time.

F# syntax in 60 seconds

Here is a very quick overview on how to read F# code for newcomers unfamiliar with the syntax.

It is obviously not very detailed but should be enough so that you can read and get the gist of the upcoming examples in this series. Don't worry if you don't understand all of it, as I will give more detailed explanations when we get to the actual code examples.

The two major differences between F# syntax and a standard C-like syntax are:

- Curly braces are not used to delimit blocks of code. Instead, indentation is used (Python is similar this way).
- Whitespace is used to separate parameters rather than commas.

Some people find the F# syntax off-putting. If you are one of them, consider this quote:

> "Optimising your notation to not confuse people in the first 10 minutes of seeing it but to hinder readability ever after is a really bad mistake." (David MacIver, via a post about Scala syntax).

Personally, I think that the F# syntax is very clear and straightforward when you get used to it. In many ways, it is simpler than the C# syntax, with fewer keywords and special cases.

The example code below is a simple F# script that demonstrates most of the concepts that you need on a regular basis.

I would encourage you to test this code interactively and play with it a bit! Either:

- Type this into a F# script file (with .fsx extension) and send it to the interactive window. See the "installing and using F#" page for details.
- Alternatively, try running this code in the interactive window. Remember to always use `;;` at the end to tell the interpreter that you are done entering and ready to evaluate.

```
// single line comments use a double slash
(* multi line comments use (* . . . *) pair

-end of multi line comment- *)

// ======== "Variables" (but not really) ==========
// The "let" keyword defines an (immutable) value
let myInt = 5
let myFloat = 3.14
let myString = "hello"    //note that no types needed
```

```
// ======== Lists ============
let twoToFive = [2;3;4;5]        // Square brackets create a list with
                                 // semicolon delimiters.
let oneToFive = 1 :: twoToFive   // :: creates list with new 1st element
// The result is [1;2;3;4;5]
let zeroToFive = [0;1] @ twoToFive   // @ concats two lists

// IMPORTANT: commas are never used as delimiters, only semicolons!

// ======== Functions ========
// The "let" keyword also defines a named function.
let square x = x * x          // Note that no parens are used.
square 3                      // Now run the function. Again, no parens.

let add x y = x + y           // don't use add (x,y)! It means something
                              // completely different.
add 2 3                       // Now run the function.

// to define a multiline function, just use indents. No semicolons needed.
let evens list =
   let isEven x = x%2 = 0     // Define "isEven" as an inner ("nested") function
   List.filter isEven list    // List.filter is a library function
                              // with two parameters: a boolean function
                              // and a list to work on

evens oneToFive               // Now run the function

// You can use parens to clarify precedence. In this example,
// do "map" first, with two args, then do "sum" on the result.
// Without the parens, "List.map" would be passed as an arg to List.sum
let sumOfSquaresTo100 =
   List.sum ( List.map square [1..100] )

// You can pipe the output of one operation to the next using "|>"
// Here is the same sumOfSquares function written using pipes
let sumOfSquaresTo100piped =
   [1..100] |> List.map square |> List.sum  // "square" was defined earlier

// you can define lambdas (anonymous functions) using the "fun" keyword
let sumOfSquaresTo100withFun =
   [1..100] |> List.map (fun x->x*x) |> List.sum

// In F# returns are implicit -- no "return" needed. A function always
// returns the value of the last expression used.

// ======== Pattern Matching ========
// Match..with.. is a supercharged case/switch statement.
let simplePatternMatch =
   let x = "a"
   match x with
    | "a" -> printfn "x is a"
    | "b" -> printfn "x is b"
    | _ -> printfn "x is something else"   // underscore matches anything
```

```
// Some(..) and None are roughly analogous to Nullable wrappers
let validValue = Some(99)
let invalidValue = None

// In this example, match..with matches the "Some" and the "None",
// and also unpacks the value in the "Some" at the same time.
let optionPatternMatch input =
   match input with
    | Some i -> printfn "input is an int=%d" i
    | None -> printfn "input is missing"

optionPatternMatch validValue
optionPatternMatch invalidValue

// ========= Complex Data Types =========

// Tuple types are pairs, triples, etc. Tuples use commas.
let twoTuple = 1,2
let threeTuple = "a",2,true

// Record types have named fields. Semicolons are separators.
type Person = {First:string; Last:string}
let person1 = {First="john"; Last="Doe"}

// Union types have choices. Vertical bars are separators.
type Temp =
    | DegreesC of float
    | DegreesF of float
let temp = DegreesF 98.6

// Types can be combined recursively in complex ways.
// E.g. here is a union type that contains a list of the same type:
type Employee =
  | Worker of Person
  | Manager of Employee list
let jdoe = {First="John";Last="Doe"}
let worker = Worker jdoe

// ========= Printing =========
// The printf/printfn functions are similar to the
// Console.Write/WriteLine functions in C#.
printfn "Printing an int %i, a float %f, a bool %b" 1 2.0 true
printfn "A string %s, and something generic %A" "hello" [1;2;3;4]

// all complex types have pretty printing built in
printfn "twoTuple=%A,\nPerson=%A,\nTemp=%A,\nEmployee=%A"
         twoTuple person1 temp worker

// There are also sprintf/sprintfn functions for formatting data
// into a string, similar to String.Format.
```

And with that, let's start by comparing some simple F# code with the equivalent C# code.

Functional languages are very different from standard imperative languages, and can be quite tricky to get the hang of initially. This page offers some tips on how to learn F# effectively.

Approach learning F# as a beginner

If you have experience in languages such as C# and Java, you have probably found that you can get a pretty good understanding of source code written in other similar languages, even if you aren't familiar with the keywords or the libraries. This is because all imperative languages use the same way of thinking, and experience in one language can be easily transferred to another.

If you are like many people, your standard approach to learning a new programming language is to find out how to implement concepts you are already familiar with. You might ask "how do I assign a variable?" or "how do I do a loop?", and with these answers be able to do some basic programming quite quickly.

When learning F#, **you should not try to bring your old imperative concepts with you**. In a pure functional language there are no variables, there are no loops, and there are no objects!

Yes, F# is a hybrid language and does support these concepts. But you will learn much faster if you start with a beginners mind.

Change the way you think

It is important to understand that functional programming is not just a stylistic difference; it is a completely different way of thinking about programming, in the way that truly object-oriented programming (in Smalltalk say) is also a different way of thinking from a traditional imperative language such as C.

F# does allow non-functional styles, and it is tempting to retain the habits you already are familiar with. You could just use F# in a non-functional way without really changing your mindset, and not realize what you are missing. To get the most out of F#, and to be fluent and comfortable with functional programming in general, it is critical that you think functionally, not imperatively.

By far the most important thing you can do is to take the time and effort to understand exactly how F# works, especially the core concepts involving functions and the type system. So please read and reread the series "thinking functionally" and "understanding F# types",

play with the examples, and get comfortable with the ideas before you try to start doing serious coding. If you don't understand how functions and types work, then you will have a hard time being productive.

Dos and Don'ts

Here is a list of dos and don'ts that will encourage you to think functionally. These will be hard at first, but just like learning a foreign language, you have to dive in and force yourself to speak like the locals.

- Don't use the `mutable` keyword **at all** as a beginner. Coding complex functions without the crutch of mutable state will really force you to understand the functional paradigm.
- Don't use `for` loops or `if-then-else`. Use pattern matching for testing booleans and recursing through lists.
- Don't use "dot notation". Instead of "dotting into" objects, try to use functions for everything. That is, write `String.length "hello"` rather than `"hello".Length`. It might seem like extra work, but this way of working is essential when using pipes and higher order functions like `List.map`. And don't write your own methods either! See this post for details.
- As a corollary, don't create classes. Use only the pure F# types such as tuples, records and unions.
- Don't use the debugger. If you have relied on the debugger to find and fix incorrect code, you will get a nasty shock. In F#, you will probably not get that far, because the compiler is so much stricter in many ways. And of course, there is no tool to "debug" the compiler and step through its processing. The best tool for debugging compiler errors is your brain, and F# forces you to use it!

On the other hand:

- Do create lots of "little types", especially union types. They are lightweight and easy, and their use will help document your domain model and ensure correctness.
- Do understand the `list` and `seq` types and their associated library modules. Functions like `List.fold` and `List.map` are very powerful. Once you understand how to use them, you will be well on your way to understanding higher order functions in general.
- Once you understand the collection modules, try to avoid recursion. Recursion can be error prone, and it can be hard to make sure that it is properly tail-recursive. When you use `List.fold`, you can never have that problem.
- Do use pipe (`|>`) and composition (`>>`) as much as you can. This style is much more idiomatic than nested function calls like `f(g(x))`
- Do understand how partial application works, and try to become comfortable with point-

free (tacit) style.

- Do develop code incrementally, using the interactive window to test code fragments. If you blindly create lots of code and then try to compile it all at once, you may end up with many painful and hard-to-debug compilation errors.

Troubleshooting

There are a number of extremely common errors that beginners make, and if you are frustrated about getting your code to compile, please read the "troubleshooting F#" page.

As the saying goes, "if it compiles, it's correct", but it can be extremely frustrating just trying to get the code to compile at all! So this page is devoted to helping you troubleshoot your F# code.

I will first present some general advice on troubleshooting and some of the most common errors that beginners make. After that, I will describe each of the common error messages in detail, and give examples of how they can occur and how to correct them.

(Jump to the error numbers)

General guidelines for troubleshooting

By far the most important thing you can do is to take the time and effort to understand exactly how F# works, especially the core concepts involving functions and the type system. So please read and reread the series "thinking functionally" and "understanding F# types", play with the examples, and get comfortable with the ideas before you try to start doing serious coding. If you don't understand how functions and types work, then the compiler errors will not make any sense.

If you are coming from an imperative language such as C#, you may have developed some bad habits by relying on the debugger to find and fix incorrect code. In F#, you will probably not get that far, because the compiler is so much stricter in many ways. And of course, there is no tool to "debug" the compiler and step through its processing. The best tool for debugging compiler errors is your brain, and F# forces you to use it!

Nevertheless, there are a number of extremely common errors that beginners make, and I will quickly go through them.

Don't use parentheses when calling a function

In F#, whitespace is the standard separator for function parameters. You will rarely need to use parentheses, and in particular, do not use parentheses when calling a function.

```
let add x y = x + y
let result = add (1 2)  //wrong
    // error FS0003: This value is not a function and cannot be applied
let result = add 1 2    //correct
```

Don't mix up tuples with multiple parameters

If it has a comma, it is a tuple. And a tuple is one object not two. So you will get errors about passing the wrong type of parameter, or too few parameters.

```
addTwoParams (1,2)  // trying to pass a single tuple rather than two args
   // error FS0001: This expression was expected to have type
   //               int but here has type 'a * 'b
```

The compiler treats `(1,2)` as a generic tuple, which it attempts to pass to " `addTwoParams` ". Then it complains that the first parameter of `addTwoParams` is an int, and we're trying to pass a tuple.

If you attempt to pass *two* arguments to a function expecting *one* tuple, you will get another obscure error.

```
addTuple 1 2   // trying to pass two args rather than one tuple
  // error FS0003: This value is not a function and cannot be applied
```

Watch out for too few or too many arguments

The F# compiler will not complain if you pass too few arguments to a function (in fact "partial application" is an important feature), but if you don't understand what is going on, you will often get strange "type mismatch" errors later.

Similarly the error for having too many arguments is typically "This value is not a function" rather than a more straightforward error.

The "printf" family of functions is very strict in this respect. The argument count must be exact.

This is a very important topic ? it is critical that you understand how partial application works. See the series "thinking functionally" for a more detailed discussion.

Use semicolons for list separators

In the few places where F# needs an explicit separator character, such as lists and records, the semicolon is used. Commas are never used. (Like a broken record, I will remind you that commas are for tuples).

```
let list1 = [1,2,3]    // wrong! This is a ONE-element list containing
                       // a three-element tuple
let list1 = [1;2;3]    // correct

type Customer = {Name:string, Address: string}  // wrong
type Customer = {Name:string; Address: string}  // correct
```

Don't use ! for not or != for not-equal

The exclamation point symbol is not the "NOT" operator. It is the deferencing operator for mutable references. If you use it by mistake, you will get the following error:

```
let y = true
let z = !y
// => error FS0001: This expression was expected to have
//    type 'a ref but here has type bool
```

The correct construction is to use the "not" keyword. Think SQL or VB syntax rather than C syntax.

```
let y = true
let z = not y       //correct
```

And for "not equal", use "<>", again like SQL or VB.

```
let z = 1 <> 2      //correct
```

Don't use = for assignment

If you are using mutable values, the assignment operation is written " `<-` ". If you use the equals symbol you might not even get an error, just an unexpected result.

```
let mutable x = 1
x = x + 1          // returns false. x is not equal to x+1
x <- x + 1         // assigns x+1 to x
```

Watch out for hidden tab characters

The indenting rules are very straightforward, and it is easy to get the hang of them. But you are not allowed to use tabs, only spaces.

```
let add x y =
{tab}x + y
// => error FS1161: TABs are not allowed in F# code
```

Be sure to set your editor to convert tabs to spaces. And watch out if you are pasting code in from elsewhere. If you do run into persistent problems with a bit of code, try removing the whitespace and re-adding it.

Don't mistake simple values for function values

If you are trying to create a function pointer or delegate, watch out that you don't accidentally create a simple value that has already been evaluated.

If you want a parameterless function that you can reuse, you will need to explicitly pass a unit parameter, or define it as a lambda.

```
let reader = new System.IO.StringReader("hello")
let nextLineFn   =  reader.ReadLine()  //wrong
let nextLineFn() =  reader.ReadLine()  //correct
let nextLineFn   =  fun() -> reader.ReadLine()  //correct

let r = new System.Random()
let randomFn   =  r.Next()  //wrong
let randomFn() =  r.Next()  //correct
let randomFn   =  fun () -> r.Next()  //correct
```

See the series "thinking functionally" for more discussion of parameterless functions.

Tips for troubleshooting "not enough information" errors

The F# compiler is currently a one-pass left-to-right compiler, and so type information later in the program is unavailable to the compiler if it hasn't been parsed yet.

A number of errors can be caused by this, such as "FS0072: Lookup on object of indeterminate type" and "FS0041: A unique overload for could not be determined". The suggested fixes for each of these specific cases are described below, but there are some general principles that can help if the compiler is complaining about missing types or not enough information. These guidelines are:

- Define things before they are used (this includes making sure the files are compiled in the right order)
- Put the things that have "known types" earlier than things that have "unknown types". In particular, you might be able reorder pipes and similar chained functions so that the typed objects come first.
- Annotate as needed. One common trick is to add annotations until everything works, and then take them away one by one until you have the minimum needed.

Do try to avoid annotating if possible. Not only is it not aesthetically pleasing, but it makes the code more brittle. It is a lot easier to change types if there are no explicit dependencies on them.

F# compiler errors

A listing of common errors, ordered by error number

Here is a list of the major errors that seem to me worth documenting. I have not documented any errors that are self explanatory, only those that seem obscure to beginners.

I will continue to add to the list in the future, and I welcome any suggestions for additions.

- FS0001: The type 'X' does not match the type 'Y'
- FS0003: This value is not a function and cannot be applied
- FS0008: This runtime coercion or type test involves an indeterminate type
- FS0010: Unexpected identifier in binding
- FS0010: Incomplete structured construct
- FS0013: The static coercion from type X to Y involves an indeterminate type
- FS0020: This expression should have type 'unit'
- FS0030: Value restriction
- FS0035: This construct is deprecated
- FS0039: The field, constructor or member X is not defined
- FS0041: A unique overload for could not be determined
- FS0049: Uppercase variable identifiers should not generally be used in patterns
- FS0072: Lookup on object of indeterminate type
- FS0588: Block following this 'let' is unfinished

FS0001: The type 'X' does not match the type 'Y'

This is probably the most common error you will run into. It can manifest itself in a wide variety of contexts, so I have grouped the most common problems together with examples and fixes. Do pay attention to the error message, as it is normally quite explicit about what the problem is.

Error message	Possible causes
The type 'float' does not match the type 'int'	A. Can't mix floats and ints
The type 'int' does not support any operators named 'DivideByInt'	A. Can't mix floats and ints.
The type 'X' is not compatible with any of the types	B. Using the wrong numeric type.
This type (function type) does not match the type (simple type). Note: function types have a arrow in them, like `'a -> 'b`.	C. Passing too many arguments to a function.
This expression was expected to have (function type) but here has (simple type)	C. Passing too many arguments to a function.
This expression was expected to have (N part function) but here has (N-1 part function)	C. Passing too many arguments to a function.
This expression was expected to have (simple type) but here has (function type)	D. Passing too few arguments to a function.
This expression was expected to have (type) but here has (other type)	E. Straightforward type mismatch. F. Inconsistent returns in branches or matches. G. Watch out for type inference effects buried in a function.
Type mismatch. Expecting a (simple type) but given a (tuple type). Note: tuple types have a star in them, like `'a * 'b`.	H. Have you used a comma instead of space or semicolon?
Type mismatch. Expecting a (tuple type) but given a (different tuple type).	I. Tuples must be the same type to be compared.
This expression was expected to have type 'a ref but here has type X	J. Don't use ! as the "not" operator.
The type (type) does not match the type (other type)	K. Operator precedence (especially functions and pipes).
This expression was expected to have type (monadic type) but here has type 'b * 'c	L. let! error in computation expressions.

A. Can't mix ints and floats

Unlike C# and most imperative languages, ints and floats cannot be mixed in expressions. You will get a type error if you attempt this:

```
1 + 2.0  //wrong
   // => error FS0001: The type 'float' does not match the type 'int'
```

The fix is to cast the int into a `float` first:

```
float 1 + 2.0  //correct
```

This issue can also manifest itself in library functions and other places. For example, you cannot do " `average` " on a list of ints.

```
[1..10] |> List.average   // wrong
   // => error FS0001: The type 'int' does not support any
   //    operators named 'DivideByInt'
```

You must cast each int to a float first, as shown below:

```
[1..10] |> List.map float |> List.average  //correct
[1..10] |> List.averageBy float  //correct (uses averageBy)
```

B. Using the wrong numeric type

You will get a "not compatible" error when a numeric cast failed.

```
printfn "hello %i" 1.0  // should be a int not a float
  // error FS0001: The type 'float' is not compatible
  //               with any of the types byte,int16,int32...
```

One possible fix is to cast it if appropriate.

```
printfn "hello %i" (int 1.0)
```

C. Passing too many arguments to a function

```
let add x y = x + y
let result = add 1 2 3
// ==> error FS0001: The type ''a -> 'b' does not match the type 'int'
```

The clue is in the error.

The fix is to remove one of the arguments!

Similar errors are caused by passing too many arguments to `printf`.

```
printfn "hello" 42
// ==> error FS0001: This expression was expected to have type 'a -> 'b
//                   but here has type unit

printfn "hello %i" 42 43
// ==> Error FS0001: Type mismatch. Expecting a 'a -> 'b -> 'c
//                   but given a 'a -> unit

printfn "hello %i %i" 42 43 44
// ==> Error FS0001: Type mismatch. Expecting a  'a -> 'b -> 'c -> 'd
//                   but given a 'a -> 'b -> unit
```

D. Passing too few arguments to a function

If you do not pass enough arguments to a function, you will get a partial application. When you later use it, you get an error because it is not a simple type.

```
let reader = new System.IO.StringReader("hello");

let line = reader.ReadLine        //wrong but compiler doesn't complain
printfn "The line is %s" line     //compiler error here!
// ==> error FS0001: This expression was expected to have type string
//                   but here has type unit -> string
```

This is particularly common for some .NET library functions that expect a unit parameter, such as `ReadLine` above.

The fix is to pass the correct number of parameters. Check the type of the result value to make sure that it is indeed a simple type. In the `ReadLine` case, the fix is to pass a `()` argument.

```
let line = reader.ReadLine()      //correct
printfn "The line is %s" line     //no compiler error
```

E. Straightforward type mismatch

The simplest case is that you have the wrong type, or you are using the wrong type in a print format string.

```
printfn "hello %s" 1.0
// => error FS0001: This expression was expected to have type string
//                  but here has type float
```

F. Inconsistent return types in branches or matches

A common mistake is that if you have a branch or match expression, then every branch MUST return the same type. If not, you will get a type error.

```
let f x =
  if x > 1 then "hello"
  else 42
// => error FS0001: This expression was expected to have type string
//                  but here has type int
```

```
let g x =
  match x with
  | 1 -> "hello"
  | _ -> 42
// error FS0001: This expression was expected to have type
//               string but here has type int
```

Obviously, the straightforward fix is to make each branch return the same type.

```
let f x =
  if x > 1 then "hello"
  else "42"

let g x =
  match x with
  | 1 -> "hello"
  | _ -> "42"
```

Remember that if an "else" branch is missing, it is assumed to return unit, so the "true" branch must also return unit.

```
let f x =
  if x > 1 then "hello"
// error FS0001: This expression was expected to have type
//               unit but here has type string
```

If both branches cannot return the same type, you may need to create a new union type that can contain both types.

```
type StringOrInt = | S of string | I of int  // new union type
let f x =
  if x > 1 then S "hello"
  else I 42
```

G. Watch out for type inference effects buried in a function

A function may cause an unexpected type inference that ripples around your code. For example, in the following, the innocent print format string accidentally causes `doSomething` to expect a string.

```
let doSomething x =
   // do something
   printfn "x is %s" x
   // do something more

doSomething 1
// => error FS0001: This expression was expected to have type string
//    but here has type int
```

The fix is to check the function signatures and drill down until you find the guilty party. Also, use the most generic types possible, and avoid type annotations if possible.

H. Have you used a comma instead of space or semicolon?

If you are new to F#, you might accidentally use a comma instead of spaces to separate function arguments:

```
// define a two parameter function
let add x y = x + 1

add(x,y)   // FS0001: This expression was expected to have
           // type int but here has type  'a * 'b
```

The fix is: don't use a comma!

```
add x y    // OK
```

One area where commas *are* used is when calling .NET library functions. These all take tuples as arguments, so the comma form is correct. In fact, these calls look just the same as they would from C#:

```
// correct
System.String.Compare("a","b")

// incorrect
System.String.Compare "a" "b"
```

I. Tuples must be the same type to be compared or pattern matched

Tuples with different types cannot be compared. Trying to compare a tuple of type `int * int`, with a tuple of type `int * string` results in an error:

```
let  t1 = (0, 1)
let  t2 = (0, "hello")
t1 = t2
// => error FS0001: Type mismatch. Expecting a int * int
//    but given a int * string
//    The type 'int' does not match the type 'string'
```

And the length must be the same:

```
let  t1 = (0, 1)
let  t2 = (0, 1, "hello")
t1 = t2
// => error FS0001: Type mismatch. Expecting a int * int
//    but given a int * int * string
//    The tuples have differing lengths of 2 and 3
```

You can get the same issue when pattern matching tuples during binding:

```
let x,y = 1,2,3
// => error FS0001: Type mismatch. Expecting a 'a * 'b
//                  but given a 'a * 'b * 'c
//                  The tuples have differing lengths of 2 and 3

let f (x,y) = x + y
let z = (1,"hello")
let result = f z
// => error FS0001: Type mismatch. Expecting a int * int
//                  but given a int * string
//                  The type 'int' does not match the type 'string'
```

J. Don't use ! as the "not" operator

If you use `!` as a "not" operator, you will get a type error mentioning the word "ref".

```
let y = true
let z = !y     //wrong
// => error FS0001: This expression was expected to have
//    type 'a ref but here has type bool
```

The fix is to use the "not" keyword instead.

```
let y = true
let z = not y   //correct
```

K. Operator precedence (especially functions and pipes)

If you mix up operator precedence, you may get type errors. Generally, function application is highest precedence compared to other operators, so you get an error in the case below:

```
String.length "hello" + "world"
   // => error FS0001:  The type 'string' does not match the type 'int'

// what is really happening
(String.length "hello") + "world"
```

The fix is to use parentheses.

```
String.length ("hello" + "world")  // corrected
```

Conversely, the pipe operator is low precedence compared to other operators.

```
let result = 42 + [1..10] |> List.sum
 // => => error FS0001:  The type ''a list' does not match the type 'int'

// what is really happening
let result = (42 + [1..10]) |> List.sum
```

Again, the fix is to use parentheses.

```
let result = 42 + ([1..10] |> List.sum)
```

L. let! error in computation expressions (monads)

Here is a simple computation expression:

```
type Wrapper<'a> = Wrapped of 'a

type wrapBuilder() =
    member this.Bind (wrapper:Wrapper<'a>) (func:'a->Wrapper<'b>) =
        match wrapper with
        | Wrapped(innerThing) -> func innerThing

    member this.Return innerThing =
        Wrapped(innerThing)

let wrap = new wrapBuilder()
```

However, if you try to use it, you get an error.

```
wrap {
    let! x1 = Wrapped(1)   // <== error here
    let! y1 = Wrapped(2)
    let z1 = x + y
    return z
    }
// error FS0001: This expression was expected to have type Wrapper<'a>
//                but here has type 'b * 'c
```

The reason is that " `Bind` " expects a tuple `(wrapper,func)` , not two parameters. (Check the signature for bind in the F# documentation).

The fix is to change the bind function to accept a tuple as its (single) parameter.

```
type wrapBuilder() =
    member this.Bind (wrapper:Wrapper<'a>, func:'a->Wrapper<'b>) =
        match wrapper with
        | Wrapped(innerThing) -> func innerThing
```

FS0003: This value is not a function and cannot be applied

This error typically occurs when passing too many arguments to a function.

```
let add1 x = x + 1
let x = add1 2 3
// ==>   error FS0003: This value is not a function and cannot be applied
```

It can also occur when you do operator overloading, but the operators cannot be used as prefix or infix.

```
let (!!) x y = x + y
(!!) 1 2              // ok
1 !! 2                // failed !! cannot be used as an infix operator
// error FS0003: This value is not a function and cannot be applied
```

FS0008: This runtime coercion or type test involves an indeterminate type

You will often see this when attempting to use " `:?` " operator to match on a type.

```
let detectType v =
    match v with
        | :? int -> printfn "this is an int"
        | _ -> printfn "something else"
// error FS0008: This runtime coercion or type test from type 'a to int
// involves an indeterminate type based on information prior to this program point.
// Runtime type tests are not allowed on some types. Further type annotations are need
ed.
```

The message tells you the problem: "runtime type tests are not allowed on some types".

The answer is to "box" the value which forces it into a reference type, and then you can type check it:

```
let detectTypeBoxed v =
    match box v with      // used "box v"
        | :? int -> printfn "this is an int"
        | _ -> printfn "something else"

//test
detectTypeBoxed 1
detectTypeBoxed 3.14
```

FS0010: Unexpected identifier in binding

Typically caused by breaking the "offside" rule for aligning expressions in a block.

```
//3456789
let f =
  let x=1     // offside line is at column 3
   x+1        // oops! don't start at column 4
              // error FS0010: Unexpected identifier in binding
```

The fix is to align the code correctly!

See also FS0588: Block following this 'let' is unfinished for another issue caused by alignment.

FS0010: Incomplete structured construct

Often occurs if you are missing parentheses from a class constructor:

```
type Something() =
   let field = ()

let x1 = new Something     // Error FS0010
let x2 = new Something()   // OK!
```

Can also occur if you forgot to put parentheses around an operator:

```
// define new operator
let (|+) a = -a

|+ 1    // error FS0010:
        // Unexpected infix operator

(|+) 1  // with parentheses -- OK!
```

Can also occur if you are missing one side of an infix operator:

```
|| true  // error FS0010: Unexpected symbol '||'
false || true  // OK
```

Can also occur if you attempt to send a namespace definition to F# interactive. The interactive console does not allow namespaces.

```
namespace Customer  // FS0010: Incomplete structured construct

// declare a type
type Person= {First:string; Last:string}
```

FS0013: The static coercion from type X to Y involves an indeterminate type

This is generally caused by implic

FS0020: This expression should have type 'unit'

This error is commonly found in two situations:

- Expressions that are not the last expression in the block
- Using wrong assignment operator

FS0020 with expressions that are not the last expression in the block

Only the last expression in a block can return a value. All others must return unit. So this typically occurs when you have a function in a place that is not the last function.

```
let something =
  2+2               // => FS0020: This expression should have type 'unit'
  "hello"
```

The easy fix is use `ignore`. But ask yourself why you are using a function and then throwing away the answer ? it might be a bug.

```
let something =
  2+2 |> ignore     // ok
  "hello"
```

This also occurs if you think you writing C# and you accidentally use semicolons to separate expressions:

```
// wrong
let result = 2+2; "hello";

// fixed
let result = 2+2 |> ignore; "hello";
```

FS0020 with assignment

Another variant of this error occurs when assigning to a property.

```
This expression should have type 'unit', but has type 'Y'.
```

With this error, chances are you have confused the assignment operator "`<-`" for mutable values, with the equality comparison operator "`=`".

```
// '=' versus '<-'
let add() =
    let mutable x = 1
    x = x + 1          // warning FS0020
    printfn "%d" x
```

The fix is to use the proper assignment operator.

```
// fixed
let add() =
    let mutable x = 1
    x <- x + 1
    printfn "%d" x
```

FS0030: Value restriction

This is related to F#'s automatic generalization to generic types whenever possible.

For example, given :

```
let id x = x
let compose f g x = g (f x)
let opt = None
```

F#'s type inference will cleverly figure out the generic types.

```
val id : 'a -> 'a
val compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
val opt : 'a option
```

However in some cases, the F# compiler feels that the code is ambiguous, and, even though it looks like it is guessing the type correctly, it needs you to be more specific:

```
let idMap = List.map id             // error FS0030
let blankConcat = String.concat ""  // error FS0030
```

Almost always this will be caused by trying to define a partially applied function, and almost always, the easiest fix is to explicitly add the missing parameter:

```
let idMap list = List.map id list             // OK
let blankConcat list = String.concat "" list  // OK
```

For more details see the MSDN article on "automatic generalization".

FS0035: This construct is deprecated

F# syntax has been cleaned up over the last few years, so if you are using examples from an older F# book or webpage, you may run into this. See the MSDN documentation for the correct syntax.

```
let x = 10
let rnd1 = System.Random x          // Good
let rnd2 = new System.Random(x)     // Good
let rnd3 = new System.Random x      // error FS0035
```

FS0039: The field, constructor or member X is not defined

This error is commonly found in four situations:

- The obvious case where something really isn't defined! And make sure that you don't have a typo or case mismatch either.
- Interfaces
- Recursion
- Extension methods

FS0039 with interfaces

In F# all interfaces are "explicit" implementations rather than "implicit". (Read the C# documentation on "explicit interface implementation" for an explanation of the difference).

The key point is that when a interface member is explicitly implemented, it cannot be accessed through a normal class instance, but only through an instance of the interface, so you have to cast to the interface type by using the `:>` operator.

Here's an example of a class that implements an interface:

```
type MyResource() =
   interface System.IDisposable with
       member this.Dispose() = printfn "disposed"
```

This doesn't work:

```
let x = new MyResource()
x.Dispose()  // error FS0039: The field, constructor
             // or member 'Dispose' is not defined
```

The fix is to cast the object to the interface, as below:

```
// fixed by casting to System.IDisposable
(x :> System.IDisposable).Dispose()   // OK

let y =  new MyResource() :> System.IDisposable
y.Dispose()   // OK
```

FS0039 with recursion

Here's a standard Fibonacci implementation:

```
let fib i =
   match i with
   | 1 -> 1
   | 2 -> 1
   | n -> fib(n-1) + fib(n-2)
```

Unfortunately, this will not compile:

```
Error FS0039: The value or constructor 'fib' is not defined
```

The reason is that when the compiler sees 'fib' in the body, it doesn't know about the function because it hasn't finished compiling it yet!

The fix is to use the " `rec` " keyword.

```
let rec fib i =
   match i with
   | 1 -> 1
   | 2 -> 1
   | n -> fib(n-1) + fib(n-2)
```

Note that this only applies to "`let`" functions. Member functions do not need this, because the scope rules are slightly different.

```
type FibHelper() =
    member this.fib i =
       match i with
       | 1 -> 1
       | 2 -> 1
       | n -> fib(n-1) + fib(n-2)
```

FS0039 with extension methods

If you have defined an extension method, you won't be able to use it unless the module is in scope.

Here's a simple extension to demonstrate:

```
module IntExtensions =
    type System.Int32 with
        member this.IsEven = this % 2 = 0
```

If you try to use it the extension, you get the FS0039 error:

```
let i = 2
let result = i.IsEven
    // FS0039: The field, constructor or
    // member 'IsEven' is not defined
```

The fix is just to open the `IntExtensions` module.

```
open IntExtensions // bring module into scope
let i = 2
let result = i.IsEven  // fixed!
```

FS0041: A unique overload for could not be determined

This can be caused when calling a .NET library function that has multiple overloads:

```
let streamReader filename = new System.IO.StreamReader(filename) // FS0041
```

There a number of ways to fix this. One way is to use an explicit type annotation:

```
let streamReader filename = new System.IO.StreamReader(filename:string) // OK
```

You can sometimes use a named parameter to avoid the type annotation:

```
let streamReader filename = new System.IO.StreamReader(path=filename) // OK
```

Or you can try to create intermediate objects that help the type inference, again without needing type annotations:

```
let streamReader filename =
    let fileInfo = System.IO.FileInfo(filename)
    new System.IO.StreamReader(fileInfo.FullName) // OK
```

FS0049: Uppercase variable identifiers should not generally be used in patterns

When pattern matching, be aware of a subtle difference between the pure F# union types which consist of a tag only, and a .NET Enum type.

Pure F# union type:

```
type ColorUnion = Red | Yellow
let redUnion = Red

match redUnion with
| Red -> printfn "red"     // no problem
| _ -> printfn "something else"
```

But with .NET enums you must fully qualify them:

```
type ColorEnum = Green=0 | Blue=1      // enum
let blueEnum = ColorEnum.Blue

match blueEnum with
| Blue -> printfn "blue"     // warning FS0049
| _ -> printfn "something else"
```

The fixed version:

```
match blueEnum with
| ColorEnum.Blue -> printfn "blue"
| _ -> printfn "something else"
```

FS0072: Lookup on object of indeterminate type

This occurs when "dotting into" an object whose type is unknown.

Consider the following example:

```
let stringLength x = x.Length // Error FS0072
```

The compiler does not know what type "x" is, and therefore does not know if "`Length`" is a valid method.

There a number of ways to fix this. The crudest way is to provide an explicit type annotation:

```
let stringLength (x:string) = x.Length  // OK
```

In some cases though, judicious rearrangement of the code can help. For example, the example below looks like it should work. It's obvious to a human that the `List.map` function is being applied to a list of strings, so why does `x.Length` cause an error?

```
List.map (fun x -> x.Length) ["hello"; "world"] // Error FS0072
```

The reason is that the F# compiler is currently a one-pass compiler, and so type information present later in the program cannot be used if it hasn't been parsed yet.

Yes, you can always explicitly annotate:

```
List.map (fun x:string -> x.Length) ["hello"; "world"] // OK
```

But another, more elegant way that will often fix the problem is to rearrange things so the known types come first, and the compiler can digest them before it moves to the next clause.

```
["hello"; "world"] |> List.map (fun x -> x.Length)   // OK
```

It's good practice to avoid explicit type annotations, so this approach is best, if it is feasible.

FS0588: Block following this 'let' is unfinished

Caused by outdenting an expression in a block, and thus breaking the "offside rule".

```
//3456789
let f =
  let x=1    // offside line is at column 3
 x+1         // offside! You are ahead of the ball!
             // error FS0588: Block following this
             // 'let' is unfinished
```

The fix is to align the code correctly.

See also FS0010: Unexpected identifier in binding for another issue caused by alignment.

So you're all excited about functional programming, and you've been learning F# in your spare time, and you're annoying your co-workers by ranting about how great it is, and you're itching to use it for serious stuff at work...

But then you hit a brick wall.

Your workplace has a "C# only" policy and won't let you use F#.

If you work in a typical enterprise environment, getting a new language approved will be a long drawn out process, involving persuading your teammates, the QA guys, the ops guys, your boss, your boss's boss, and the mysterious bloke down the hall who you've never talked to. I would encourage you to start that process (a helpful link for your manager), but still, you're impatient and thinking "what can I do now?"

On the other hand, perhaps you work in a flexible, easy going place, where you can do what you like.

But you're conscientious, and don't want to be one of those people who re-write some mission critical system in APL, and then vanish without trace, leaving your replacement some mind-bendingly cryptic code to maintain. No, you want to make sure that you are not doing anything that will affect your team's bus factor.

So in both these scenarios, you want to use F# at work, but you can't (or don't want to) use it for core application code.

What can you do?

Well, don't worry! This series will suggest a number of ways you can get your hands dirty with F# in a low-risk, incremental way, without affecting any mission critical code.

- Twenty six low-risk ways to use F# at work. You can start right now -- no permission needed.
- Using F# for development and devops scripts. Twenty six low-risk ways to use F# at work (part 2).
- Using F# for testing. Twenty six low-risk ways to use F# at work (part 3).
- Using F# for database related tasks. Twenty six low-risk ways to use F# at work (part 4).
- Other interesting ways of using F# at work. Twenty six low-risk ways to use F# at work (part 5).

Twenty six low-risk ways to use F# at work

So you're all excited about functional programming, and you've been learning F# in your spare time, and you're annoying your co-workers by ranting about how great it is, and you're itching to use it for serious stuff at work...

But then you hit a brick wall.

Your workplace has a "C# only" policy and won't let you use F#.

If you work in a typical enterprise environment, getting a new language approved will be a long drawn out process, involving persuading your teammates, the QA guys, the ops guys, your boss, your boss's boss, and the mysterious bloke down the hall who you've never talked to. I would encourage you to start that process (a helpful link for your manager), but still, you're impatient and thinking "what can I do now?"

On the other hand, perhaps you work in a flexible, easy going place, where you can do what you like.

But you're conscientious, and don't want to be one of those people who re-write some mission critical system in APL, and then vanish without trace, leaving your replacement some mind-bendingly cryptic code to maintain. No, you want to make sure that you are not doing anything that will affect your team's bus factor.

So in both these scenarios, you want to use F# at work, but you can't (or don't want to) use it for core application code.

What can you do?

Well, don't worry! This series of articles will suggest a number of ways you can get your hands dirty with F# in a low-risk, incremental way, without affecting any critical code.

Series contents

Here's a list of the twenty six ways so that you can go straight to any one that you find particularly interesting.

Part 1 - Using F# to explore and develop interactively

1. Use F# to explore the .NET framework interactively
2. Use F# to test your own code interactively
3. Use F# to play with webservices interactively

4. Use F# to play with UI's interactively

Part 2 - Using F# for development and devops scripts

5. Use FAKE for build and CI scripts
6. An F# script to check that a website is responding
7. An F# script to convert an RSS feed into CSV
8. An F# script that uses WMI to check the stats of a process
9. Use F# for configuring and managing the cloud

Part 3 - Using F# for testing

10. Use F# to write unit tests with readable names
11. Use F# to run unit tests programmatically
12. Use F# to learn to write unit tests in other ways
13. Use FsCheck to write better unit tests
14. Use FsCheck to create random dummy data
15. Use F# to create mocks
16. Use F# to do automated browser testing
17. Use F# for Behaviour Driven Development

Part 4. Using F# for database related tasks

18. Use F# to replace LINQpad
19. Use F# to unit test stored procedures
20. Use FsCheck to generate random database records
21. Use F# to do simple ETL
22. Use F# to generate SQL Agent scripts

Part 5: Other interesting ways of using F#

23. Use F# for parsing
24. Use F# for diagramming and visualization
25. Use F# for accessing web-based data stores
26. Use F# for data science and machine learning

(BONUS) 27: Balance the generation schedule for the UK power station fleet

Getting started

If you're using Visual Studio, you've already got F# installed, so you're ready to go! No need to ask anyone's permission.

If you're on a Mac or Linux, you will have to a bit of work, alas (instructions for Mac and Linux).

There are two ways to use F# interactively: (1) typing in the F# interactive window directly, or (2) creating a F# script file (.FSX) and then evaluating code snippets.

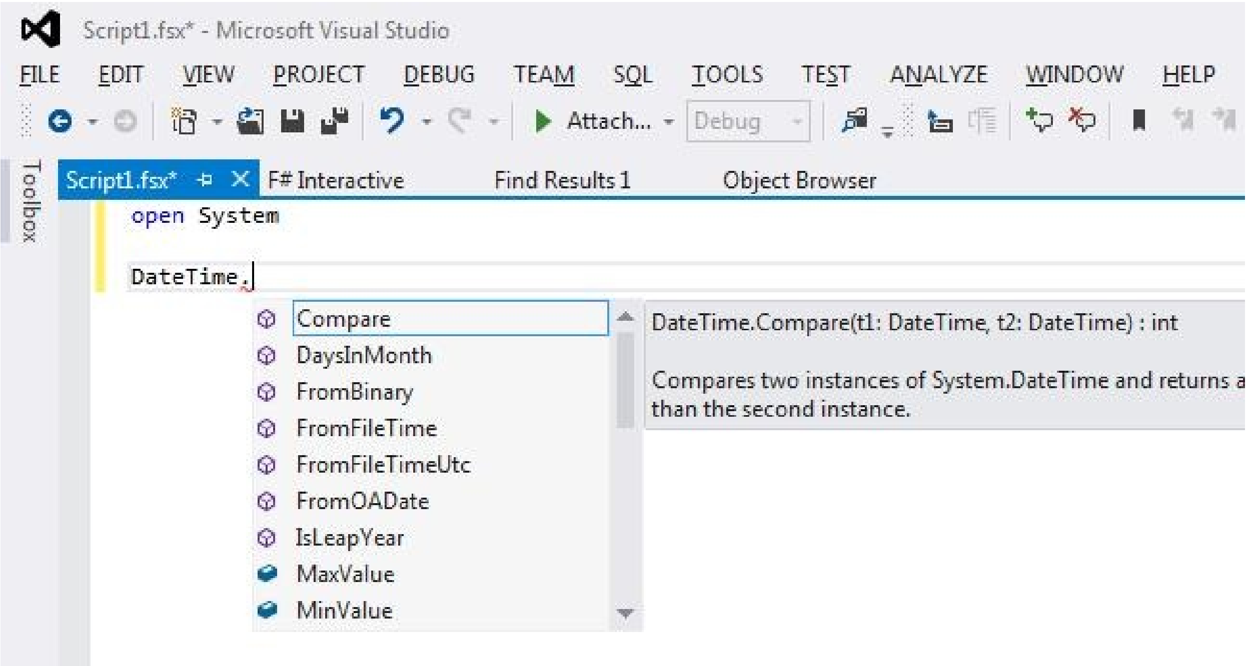
To use the F# interactive window in Visual Studio:

1. Show the window with `Menu > View > Other Windows > F# Interactive`
2. Type an expression, and use double semicolon (`;;`) to tell the interpreter you're finished.

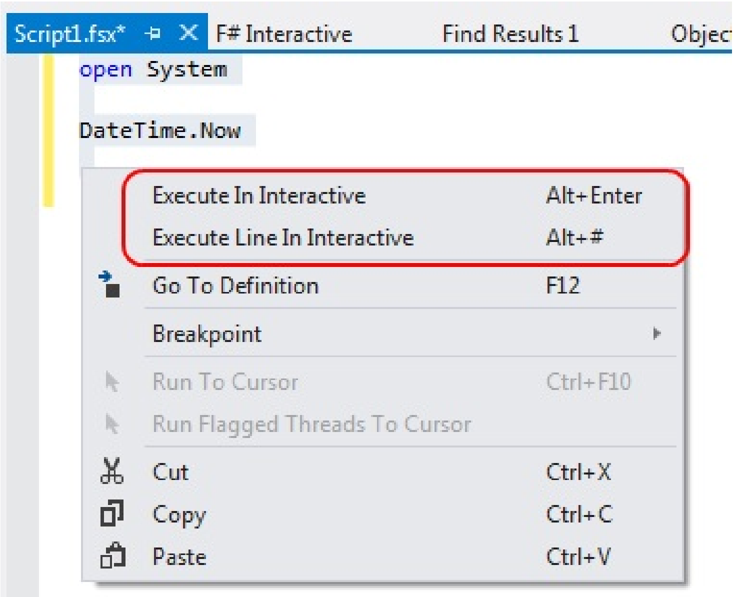
For example:

```
let x = 1
let y = 2
x + y;;
```

Personally, I prefer to create a script file (`File > New > File` then pick "F# script") and type code there, because you get auto-complete and intellisense.



To run a bit of code, just highlight and right click, or simply do `Alt+Enter` .



Working with external libraries and NuGet

Most of the code samples reference external libraries which are expected to be under the script directory.

You could download or compile these DLLs explicitly, but I think using NuGet from the command line is simpler.

1. First, you need to install Chocolately (from chocolatey.org)
2. Next install the NuGet command line using `cinst nuget.commandline`
3. Finally, go to your script directory, and install the NuGet package from the command line.
   For example, `nuget install FSharp.Data -o Packages -ExcludeVersion`
   As you see, I prefer to exclude versions from Nuget packages when using them from scripts so that I can update later without breaking existing code.

Part 1: Using F# to explore and develop interactively

The first area where F# is valuable is as a tool to interactively explore .NET libraries.

Before, in order to do this, you might have created unit tests and then stepped through them with a debugger to understand what is happening. But with F#, you don't need to do that, you can run the code directly.

Let's look at some examples.

1. Use F# to explore the .NET framework interactively

The code for this section is available on github.

When I'm coding, I often have little questions about how the .NET library works.

For example, here are some questions that I have had recently that I answered by using F# interactively:

- Have I got a custom DateTime format string correct?
- How does XML serialization handle local DateTimes vs. UTC DateTimes?
- Is `GetEnvironmentVariable` case-sensitive?

All these questions can be found in the MSDN documentation, of course, but can also answered in seconds by running some simple F# snippets, shown below.

Have I got a custom DateTime format string correct?

I want to use 24 hour clock in a custom format. I know that it's "h", but is it upper or lowercase "h"?

```
open System
DateTime.Now.ToString("yyyy-MM-dd hh:mm")  // "2014-04-18 01:08"
DateTime.Now.ToString("yyyy-MM-dd HH:mm")  // "2014-04-18 13:09"
```

How does XML serialization handle local DateTimes vs. UTC DateTimes?

How exactly, does XML serialization work with dates? Let's find out!

```
// TIP: sets the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

open System

[<CLIMutable>]
type DateSerTest = {Local:DateTime;Utc:DateTime}

let ser = new System.Xml.Serialization.XmlSerializer(typeof<DateSerTest>)

let testSerialization (dt:DateSerTest) =
    let filename = "serialization.xml"
    use fs = new IO.FileStream(filename , IO.FileMode.Create)
    ser.Serialize(fs, o=dt)
    fs.Close()
    IO.File.ReadAllText(filename) |> printfn "%s"

let d = {
    Local = DateTime.SpecifyKind(new DateTime(2014,7,4), DateTimeKind.Local)
    Utc = DateTime.SpecifyKind(new DateTime(2014,7,4), DateTimeKind.Utc)
    }

testSerialization d
```

The output is:

```
<DateSerTest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://w
ww.w3.org/2001/XMLSchema">
  <Local>2014-07-04T00:00:00+01:00</Local>
  <Utc>2014-07-04T00:00:00Z</Utc>
</DateSerTest>
```

So I can see it uses "Z" for UTC times.

Is GetEnvironmentVariable case-sensitive?

This can be answered with a simple snippet:

```
Environment.GetEnvironmentVariable "ProgramFiles" =
    Environment.GetEnvironmentVariable "PROGRAMFILES"
// answer => true
```

The answer is therefore "not case-sensitive".

2. Use F# to test your own code interactively

The code for this section is available on github.

You are not restricted to playing with the .NET libraries, of course. Sometimes it can be quite useful to test your own code.

To do this, just reference the DLL and then open the namespace as shown below.

```
// set the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

// pass in the relative path to the DLL
#r @"bin\debug\myapp.dll"

// open the namespace
open MyApp

// do something
MyApp.DoSomething()
```

WARNING: in older versions of F#, opening a reference to your DLL will lock it so that you can't compile it! In which case, before recompiling, be sure to reset the interactive session to release the lock. In newer versions of F#, the DLL is shadow-copied, and there is no lock.

3. Use F# to play with webservices interactively

The code for this section is available on github.

If you want to play with the WebAPI and Owin libraries, you don't need to create an executable -- you can do it through script alone!

There is a little bit of setup involved, as you will need a number of library DLLs to make this work.

So, assuming you have got the NuGet command line set up (see above), go to your script directory, and install the self hosting libraries via `nuget install Microsoft.AspNet.WebApi.OwinSelfHost -o Packages -ExcludeVersion`

Once these libraries are in place, you can use the code below as a skeleton for a simple WebAPI app.

```
// sets the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

// assumes nuget install Microsoft.AspNet.WebApi.OwinSelfHost has been run
// so that assemblies are available under the current directory
```

```
#r @"Packages\Owin\lib\net40\Owin.dll"
#r @"Packages\Microsoft.Owin\lib\net40\Microsoft.Owin.dll"
#r @"Packages\Microsoft.Owin.Host.HttpListener\lib\net40\Microsoft.Owin.Host.HttpListe
ner.dll"
#r @"Packages\Microsoft.Owin.Hosting\lib\net40\Microsoft.Owin.Hosting.dll"
#r @"Packages\Microsoft.AspNet.WebApi.Owin\lib\net45\System.Web.Http.Owin.dll"
#r @"Packages\Microsoft.AspNet.WebApi.Core\lib\net45\System.Web.Http.dll"
#r @"Packages\Microsoft.AspNet.WebApi.Client\lib\net45\System.Net.Http.Formatting.dll"
#r @"Packages\Newtonsoft.Json\lib\net40\Newtonsoft.Json.dll"
#r "System.Net.Http.dll"

open System
open Owin
open Microsoft.Owin
open System.Web.Http
open System.Web.Http.Dispatcher
open System.Net.Http.Formatting

module OwinSelfhostSample =

    /// a record to return
    [<CLIMutable>]
    type Greeting = { Text : string }

    /// A simple Controller
    type GreetingController() =
        inherit ApiController()

        // GET api/greeting
        member this.Get()  =
            {Text="Hello!"}

    /// Another Controller that parses URIs
    type ValuesController() =
        inherit ApiController()

        // GET api/values
        member this.Get()  =
            ["value1";"value2"]

        // GET api/values/5
        member this.Get id =
            sprintf "id is %i" id

        // POST api/values
        member this.Post ([<FromBody>]value:string) =
            ()

        // PUT api/values/5
        member this.Put(id:int, [<FromBody>]value:string) =
            ()

        // DELETE api/values/5
```

```
        member this.Delete(id:int) =
            ()

    /// A helper class to store routes, etc.
    type ApiRoute = { id : RouteParameter }

    /// IMPORTANT: When running interactively, the controllers will not be found with
error:
    /// "No type was found that matches the controller named 'XXX'."
    /// The fix is to override the ControllerResolver to use the current assembly
    type ControllerResolver() =
        inherit DefaultHttpControllerTypeResolver()

        override this.GetControllerTypes (assembliesResolver:IAssembliesResolver) =
            let t = typeof<System.Web.Http.Controllers.IHttpController>
            System.Reflection.Assembly.GetExecutingAssembly().GetTypes()
            |> Array.filter t.IsAssignableFrom
            :> Collections.Generic.ICollection<Type>

    /// A class to manage the configuration
    type MyHttpConfiguration() as this =
        inherit HttpConfiguration()

        let configureRoutes() =
            this.Routes.MapHttpRoute(
                name= "DefaultApi",
                routeTemplate= "api/{controller}/{id}",
                defaults= { id = RouteParameter.Optional }
                ) |> ignore

        let configureJsonSerialization() =
            let jsonSettings = this.Formatters.JsonFormatter.SerializerSettings
            jsonSettings.Formatting <- Newtonsoft.Json.Formatting.Indented
            jsonSettings.ContractResolver <-
                Newtonsoft.Json.Serialization.CamelCasePropertyNamesContractResolver()

        // Here is where the controllers are resolved
        let configureServices() =
            this.Services.Replace(
                typeof<IHttpControllerTypeResolver>,
                new ControllerResolver())

        do configureRoutes()
        do configureJsonSerialization()
        do configureServices()

    /// Create a startup class using the configuration
    type Startup() =

        // This code configures Web API. The Startup class is specified as a type
        // parameter in the WebApp.Start method.
        member this.Configuration (appBuilder:IAppBuilder) =
            // Configure Web API for self-host.
```

```
            let config = new MyHttpConfiguration()
            appBuilder.UseWebApi(config) |> ignore


// Start OWIN host
do
    // Create server
    let baseAddress = "http://localhost:9000/"
    use app = Microsoft.Owin.Hosting.WebApp.Start<OwinSelfhostSample.Startup>(url=base
Address)

    // Create client and make some requests to the api
    use client = new System.Net.Http.HttpClient()

    let showResponse query =
        let response = client.GetAsync(baseAddress + query).Result
        Console.WriteLine(response)
        Console.WriteLine(response.Content.ReadAsStringAsync().Result)

    showResponse "api/greeting"
    showResponse "api/values"
    showResponse "api/values/42"

    // for standalone scripts, pause so that you can test via your browser as well
    Console.ReadLine() |> ignore
```

Here's the output:

```
StatusCode: 200, ReasonPhrase: 'OK', Version: 1.1, Content: System.Net.Http.StreamCont
ent, Headers:
{
  Date: Fri, 18 Apr 2014 22:29:04 GMT
  Server: Microsoft-HTTPAPI/2.0
  Content-Length: 24
  Content-Type: application/json; charset=utf-8
}
{
  "text": "Hello!"
}
StatusCode: 200, ReasonPhrase: 'OK', Version: 1.1, Content: System.Net.Http.StreamCont
ent, Headers:
{
  Date: Fri, 18 Apr 2014 22:29:04 GMT
  Server: Microsoft-HTTPAPI/2.0
  Content-Length: 29
  Content-Type: application/json; charset=utf-8
}
[
  "value1",
  "value2"
]
StatusCode: 200, ReasonPhrase: 'OK', Version: 1.1, Content: System.Net.Http.StreamCont
ent, Headers:
{
  Date: Fri, 18 Apr 2014 22:29:04 GMT
  Server: Microsoft-HTTPAPI/2.0
  Content-Length: 10
  Content-Type: application/json; charset=utf-8
}
"id is 42"
```

This example is just to demonstrate that you can use the OWIN and WebApi libraries "out-of-the-box".

For a more F# friendly web framework, have a look at Suave or WebSharper. There is a lot more webby stuff at fsharp.org.

4. Use F# to play with UI's interactively

The code for this section is available on github.

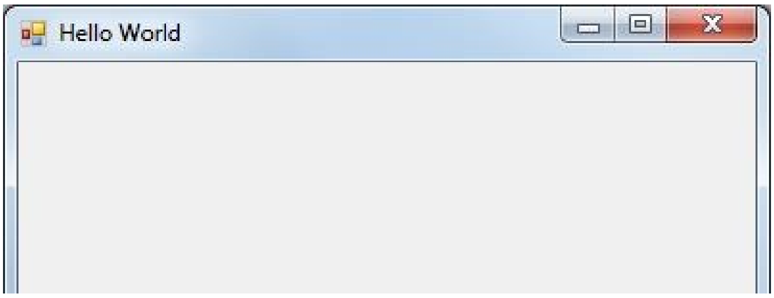
Another use for F# interactive is to play with UI's while they are running -- live!

Here's an example of developing a WinForms screen interactively.

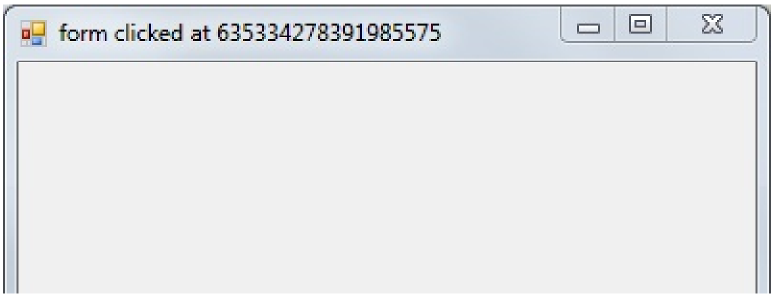
```
open System.Windows.Forms
open System.Drawing

let form = new Form(Width= 400, Height = 300, Visible = true, Text = "Hello World")
form.TopMost <- true
form.Click.Add (fun _ ->
    form.Text <- sprintf "form clicked at %i" DateTime.Now.Ticks)
form.Show()
```

Here's the window:



And here's the window after clicking, with the title bar changed:

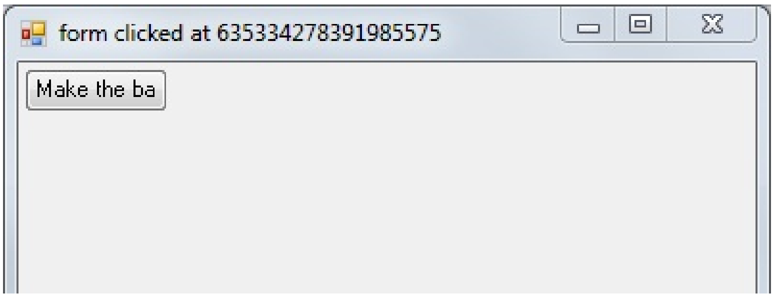


Now let's add a FlowLayoutPanel and a button.

```
let panel = new FlowLayoutPanel()
form.Controls.Add(panel)
panel.Dock = DockStyle.Fill
panel.WrapContents <- false

let greenButton = new Button()
greenButton.Text <- "Make the background color green"
greenButton.Click.Add (fun _-> form.BackColor <- Color.LightGreen)
panel.Controls.Add(greenButton)
```

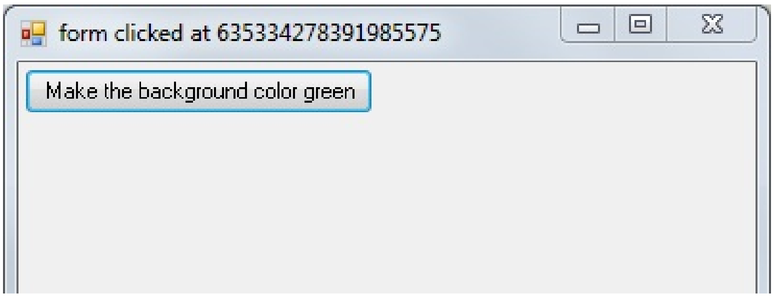
Here's the window now:



But the button is too small -- we need to set `AutoSize` to be true.

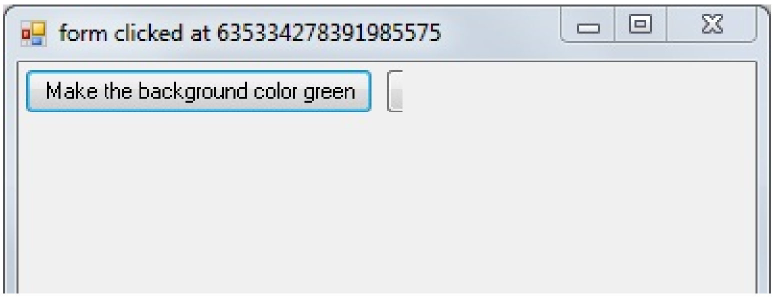
```
greenButton.AutoSize <- true
```

That's better!



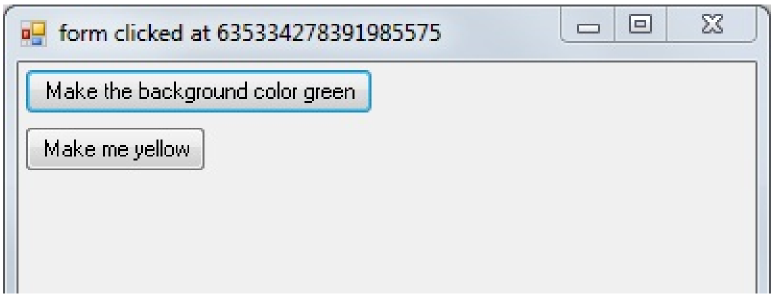
Let's add a yellow button too:

```
let yellowButton = new Button()
yellowButton.Text <- "Make me yellow"
yellowButton.AutoSize <- true
yellowButton.Click.Add (fun _-> form.BackColor <- Color.Yellow)
panel.Controls.Add(yellowButton)
```



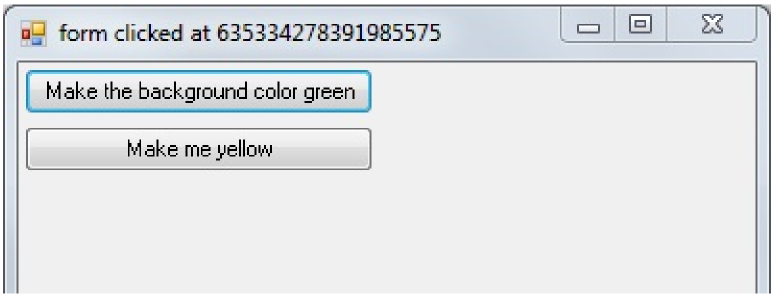
But the button is cut off, so let's change the flow direction:

```
panel.FlowDirection <- FlowDirection.TopDown
```



But now the yellow button is not the same width as the green button, which we can fix with `Dock` :

```
yellowButton.Dock <- DockStyle.Fill
```



As you can see, it is really easy to play around with layouts interactively this way. Once you're happy with the layout logic, you can convert the code back to C# for your real application.

This example is WinForms specific. For other UI frameworks the logic would be different, of course.

So that's the first four suggestions. We're not done yet! The next post will cover using F# for development and devops scripts.

Using F# for development and devops scripts

This post is a continuation of the series on low-risk ways to use F# at work. I've been suggesting a number of ways you can get your hands dirty with F# in a low-risk, incremental way, without affecting any mission critical code.

In this one, we'll talk about using F# for builds and other development and devops scripts.

If you're new to F#, you might want to read the sections on getting started and working with NuGet in the previous post.

Series contents

Here's a list of shortcuts to the twenty six ways:

Part 1 - Using F# to explore and develop interactively

1. Use F# to explore the .NET framework interactively
2. Use F# to test your own code interactively
3. Use F# to play with webservices interactively
4. Use F# to play with UI's interactively

Part 2 - Using F# for development and devops scripts

5. Use FAKE for build and CI scripts
6. An F# script to check that a website is responding
7. An F# script to convert an RSS feed into CSV
8. An F# script that uses WMI to check the stats of a process
9. Use F# for configuring and managing the cloud

Part 3 - Using F# for testing

10. Use F# to write unit tests with readable names
11. Use F# to run unit tests programmatically
12. Use F# to learn to write unit tests in other ways
13. Use FsCheck to write better unit tests
14. Use FsCheck to create random dummy data
15. Use F# to create mocks
16. Use F# to do automated browser testing
17. Use F# for Behaviour Driven Development

Part 4. Using F# for database related tasks

18. Use F# to replace LINQpad
19. Use F# to unit test stored procedures
20. Use FsCheck to generate random database records
21. Use F# to do simple ETL
22. Use F# to generate SQL Agent scripts

Part 5: Other interesting ways of using F#

23. Use F# for parsing
24. Use F# for diagramming and visualization
25. Use F# for accessing web-based data stores
26. Use F# for data science and machine learning

(BONUS) 27: Balance the generation schedule for the UK power station fleet

---

Part 2: Using F# for development and devops scripts

The next set of suggestions relates to using F# for the various scripts that revolve around development activities: builds, continuous integration, deployment, etc.

For these kinds of small tasks, you need a good scripting language with a REPL. You could use PowerShell, or ScriptCS, or even Python. But why not give F# a go?

- F# feels lightweight like Python (few or no type declarations).
- F# can access .NET libraries, both the core ones and those downloaded via NuGet.
- F# has type providers (a big advantage over PowerShell and ScriptCS) that let you easily access a wide range of data sources.
- All this in a concise, type-safe manner, with intellisense too!

Using F# in this way will allow you and your fellow developers to use F# code to solve practical problems. There shouldn't be any resistance from managers for this low-risk approach -- in the worse case you can easily switch to using a different tool.

A hidden agenda, of course, is that once your fellow developers get a chance to play with F#, they'll be hooked, and you'll be one step closer to using F# end to end!

What can you do with F# scripts?

In the next few sections we'll see three examples of F# scripts:

- An F# script to check that a website is responding
- An F# script to convert an RSS feed into CSV
- An F# script that uses WMI to check the stats of a process

But of course, you can integrate F# scripts with almost any .NET library. Here are other suggestions for utilities that can be scripted:

- Simple file copying, directory traversal, and archiving (e.g. of log files). If you're using .NET 4.5, you can use the new System.IO.Compression.ZipArchive class to do zipping and unzipping without needing a third party library.
- Doing things with JSON, either with a known format (using the JSON Type Provider) or unknown format (using the JSON parser).
- Interacting with GitHub using Octokit.
- Extracting data from, or manipulating data in, Excel. F# supports COM for doing Office automation, or you can use one of the type providers or libraries.
- Doing numerics with Math.NET.
- Web crawling, link checking, and screenscraping. The built-in async workflows and agents make this kind of "multithreaded" code very easy to write.
- Scheduling things with Quartz.NET.

If these suggestions whet your interest, and you want to use more F#, then check out the F# community projects page. It's a great source of useful libraries being written for F#, and most of them will work well with F# scripting.

Debugging F# scripts

A great thing about using F# scripts is that you don't need to create a whole project, nor launch Visual Studio.

But if you need to debug a script, and you're not in Visual Studio, what can you do? Here are some tips:

- First, you can just use tried and true printing to the console using `printfn`. I generally wrap this in a simple `log` function so that I can turn logging on or off with a flag.
- You can use the FsEye tool to inspect and watch variables in an interactive session.
- Finally, you can still use the Visual Studio debugger. The trick is to attach the debugger to the fsi.exe process, and then you can use `Debugger.Break` to halt at a certain point.

5. Use FAKE for build and CI scripts

The code for this section is available on github.

Let's start with FAKE, which is a cross platform build automation tool written in F#, analogous to Ruby's Rake.

FAKE has built-in support for git, NuGet, unit tests, Octopus Deploy, Xamarin and more, and makes it easy to develop complex scripts with dependencies.

You can even use it with TFS to avoid using XAML.

One reason to use FAKE rather than something like Rake is that you can standardize on .NET code throughout your tool chain. In theory, you could use NAnt instead, but in practice, no thanks, because XML. PSake is also a possibility, but more complicated than FAKE, I think.

You can also use FAKE to remove dependencies on a particular build server. For example, rather than using TeamCity's integration to run tests and other tasks, you might consider doing them in FAKE instead, which means you can run full builds without having TeamCity installed.

Here's an example of a very simple FAKE script, taken from a more detailed example on the FAKE site.

```
// Include Fake lib
// Assumes NuGet has been used to fetch the FAKE libraries
#r "packages/FAKE/tools/FakeLib.dll"
open Fake

// Properties
let buildDir = "./build/"

// Targets
Target "Clean" (fun _ ->
    CleanDir buildDir
)

Target "Default" (fun _ ->
    trace "Hello World from FAKE"
)

// Dependencies
"Clean"
  ==> "Default"

// start build
RunTargetOrDefault "Default"
```

The syntax takes a little getting used to, but that effort is well spent.

Some further reading on FAKE:

- Migrating to FAKE.
- Hanselman on FAKE. Many of the comments are from people who are using FAKE actively.
- A NAnt user tries out FAKE.

6. An F# script to check that a website is responding

The code for this section is available on github.

This script checks that a website is responding with a 200. This might be useful as the basis for a post-deployment smoke test, for example.

```
// Requires FSharp.Data under script directory
//    nuget install FSharp.Data -o Packages -ExcludeVersion
#r @"Packages\FSharp.Data\lib\net40\FSharp.Data.dll"
open FSharp.Data

let queryServer uri queryParams =
    try
        let response = Http.Request(uri, query=queryParams, silentHttpErrors = true)
        Some response
    with
    | :? System.Net.WebException as ex -> None

let sendAlert uri message =
    // send alert via email, say
    printfn "Error for %s. Message=%O" uri message

let checkServer (uri,queryParams) =
    match queryServer uri queryParams with
    | Some response ->
        printfn "Response for %s is %O" uri response.StatusCode
        if (response.StatusCode <> 200) then
            sendAlert uri response.StatusCode
    | None ->
        sendAlert uri "No response"

// test the sites
let google = "http://google.com", ["q","fsharp"]
let bad = "http://example.bad", []

[google;bad]
|> List.iter checkServer
```

The result is:

```
Response for http://google.com is 200
Error for http://example.bad. Message=No response
```

Note that I'm using the Http utilities code in `Fsharp.Data`, which provides a nice wrapper around `HttpClient`. More on HttpUtilities here.

7. An F# script to convert an RSS feed into CSV

The code for this section is available on github.

Here's a little script that uses the Xml type provider to parse an RSS feed (in this case, F# questions on StackOverflow) and convert it to a CSV file for later analysis.

Note that the RSS parsing code is just one line of code! Most of the code is concerned with writing the CSV. Yes, I could have used a CSV library (there are lots on NuGet) but I thought I'd leave it as is to show you how simple it is.

```
// sets the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

// Requires FSharp.Data under script directory
//    nuget install FSharp.Data -o Packages -ExcludeVersion
#r @"Packages\FSharp.Data\lib\net40\FSharp.Data.dll"
#r "System.Xml.Linq.dll"
open FSharp.Data

type Rss = XmlProvider<"http://stackoverflow.com/feeds/tag/f%23">

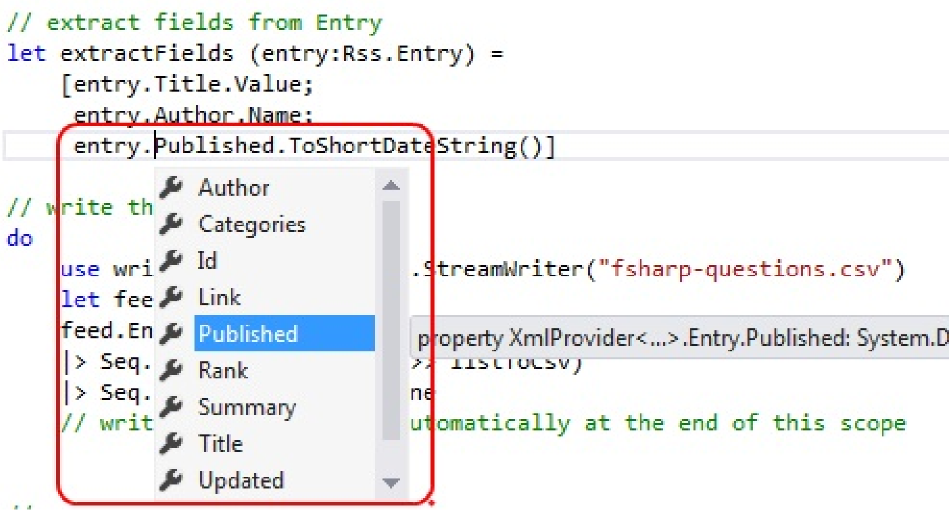
// prepare a string for writing to CSV
let prepareStr obj =
    obj.ToString()
     .Replace("\"","\"\"") // replace single with double quotes
     |> sprintf "\"%s\""   // surround with quotes

// convert a list of strings to a CSV
let listToCsv list =
    let combine s1 s2 = s1 + "," + s2
    list
    |> Seq.map prepareStr
    |> Seq.reduce combine

// extract fields from Entry
let extractFields (entry:Rss.Entry) =
    [entry.Title.Value;
     entry.Author.Name;
     entry.Published.ToShortDateString()]

// write the lines to a file
do
    use writer = new System.IO.StreamWriter("fsharp-questions.csv")
    let feed = Rss.GetSample()
    feed.Entries
    |> Seq.map (extractFields >> listToCsv)
    |> Seq.iter writer.WriteLine
    // writer will be closed automatically at the end of this scope
```

Note that the type provider generates intellisense (shown below) to show you the available properties based on the actual contents of the feed. That's very cool.



The result is something like this:

```
"Optimising F# answer for Euler #4","DropTheTable","18/04/2014"
"How to execute a function, that creates a lot of objects, in parallel?","Lawrence Woo
dman","01/04/2014"
"How to invoke a user defined function using R Type Provider","Dave","19/04/2014"
"Two types that use themselves","trn","19/04/2014"
"How does function [x] -> ... work","egerhard","19/04/2014"
```

For more on the XML type provider, see the FSharp.Data pages.

8. An F# script that uses WMI to check the stats of a process

The code for this section is available on github.

If you use Windows, being able to access WMI is very useful. Luckily there is an F# type provider for WMI that makes using it easy.

In this example, we'll get the system time and also check some stats for a process. This could be useful during and after a load test, for example.

```
// sets the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

// Requires FSharp.Management under script directory
//    nuget install FSharp.Management -o Packages -ExcludeVersion
#r @"System.Management.dll"
#r @"Packages\FSharp.Management\lib\net40\FSharp.Management.dll"
#r @"Packages\FSharp.Management\lib\net40\FSharp.Management.WMI.dll"

open FSharp.Management

// get data for the local machine
type Local = WmiProvider<"localhost">
let data = Local.GetDataContext()

// get the time and timezone on the machine
let time = data.Win32_UTCTime |> Seq.head
let tz = data.Win32_TimeZone |> Seq.head
printfn "Time=%O-%O-%O %O:%O:%O" time.Year time.Month time.Day time.Hour time.Minute t
ime.Second
printfn "Timezone=%O" tz.StandardName

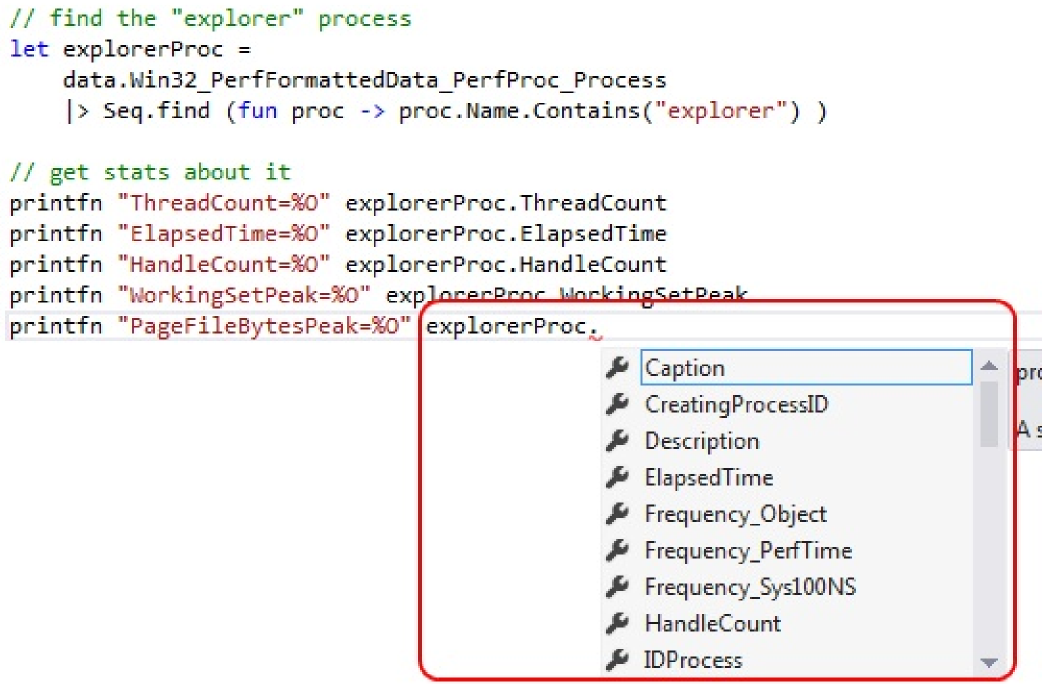
// find the "explorer" process
let explorerProc =
    data.Win32_PerfFormattedData_PerfProc_Process
    |> Seq.find (fun proc -> proc.Name.Contains("explorer") )

// get stats about it
printfn "ElapsedTime=%O" explorerProc.ElapsedTime
printfn "ThreadCount=%O" explorerProc.ThreadCount
printfn "HandleCount=%O" explorerProc.HandleCount
printfn "WorkingSetPeak=%O" explorerProc.WorkingSetPeak
printfn "PageFileBytesPeak=%O" explorerProc.PageFileBytesPeak
```

The output is something like this:

```
Time=2014-4-20 14:2:35
Timezone=GMT Standard Time
ElapsedTime=2761906
ThreadCount=67
HandleCount=3700
WorkingSetPeak=168607744
PageFileBytesPeak=312565760
```

Again, using a type provider means that you get intellisense (shown below). Very useful for the hundreds of WMI options.



More on the WMI type provider here.

9. Use F# for configuring and managing the cloud

One area which deserves special mention is using F# for configuring and managing cloud services. The cloud page at fsharp.org has many helpful links.

For simple scripting, Fog is a nice wrapper for Azure.

So for example, to upload a blob, the code is as simple as this:

```
UploadBlob "testcontainer" "testblob" "This is a test" |> ignore
```

or to add and receive messages:

```
AddMessage "testqueue" "This is a test message" |> ignore

let result = GetMessages "testqueue" 20 5
for m in result do
    DeleteMessage "testqueue" m
```

What's especially nice about using F# for this is that you can do it in micro scripts -- you don't need any heavy tooling.

Summary

I hope you found these suggestions useful. Let me know in the comments if you apply them in practice.

Next up: using F# for testing.

Using F# for testing

This post is a continuation of the previous series on low-risk and incremental ways to use F# at work -- how can you get your hands dirty with F# in a low-risk, incremental way, without affecting any mission critical code?

In this one, we'll talk about using F# for testing.

Series contents

Before moving on to the content of the post, here's the full list of the twenty six ways:

Part 1 - Using F# to explore and develop interactively

1. Use F# to explore the .NET framework interactively
2. Use F# to test your own code interactively
3. Use F# to play with webservices interactively
4. Use F# to play with UI's interactively

Part 2 - Using F# for development and devops scripts

5. Use FAKE for build and CI scripts
6. An F# script to check that a website is responding
7. An F# script to convert an RSS feed into CSV
8. An F# script that uses WMI to check the stats of a process
9. Use F# for configuring and managing the cloud

Part 3 - Using F# for testing

10. Use F# to write unit tests with readable names
11. Use F# to run unit tests programmatically
12. Use F# to learn to write unit tests in other ways
13. Use FsCheck to write better unit tests
14. Use FsCheck to create random dummy data
15. Use F# to create mocks
16. Use F# to do automated browser testing
17. Use F# for Behaviour Driven Development

Part 4. Using F# for database related tasks

18. Use F# to replace LINQpad
19. Use F# to unit test stored procedures
20. Use FsCheck to generate random database records
21. Use F# to do simple ETL
22. Use F# to generate SQL Agent scripts

Part 5: Other interesting ways of using F#

23. Use F# for parsing
24. Use F# for diagramming and visualization
25. Use F# for accessing web-based data stores
26. Use F# for data science and machine learning
(BONUS) 27: Balance the generation schedule for the UK power station fleet

---

Part 3 - Using F# for testing

If you want to start writing useful code in F# without touching core code, writing tests is a great way to start.

Not only does F# have a more compact syntax, it also has many nice features, such as the "double backtick" syntax, that make test names much more readable.

As with all of the suggestions in this series, I think this is a low risk option. Test methods tend to be short, so almost anyone will be able to read them without having to understand F# deeply. In the worst-case, you can easily port them back to C#.

10. Use F# to write unit tests with readable names

The code for this section is available on github.

Just like C#, F# can be used to write standard unit tests using the standard frameworks like NUnit, MsUnit, xUnit, etc.

Here's an example of a test class written for use with NUnit.

```
[<TestFixture>]
type TestClass() =

    [<Test>]
    member this.When2IsAddedTo2Expect4() =
        Assert.AreEqual(4, 2+2)
```

As you can see, there's a class with the `TestFixture` attribute, and a public void method with the `Test` attribute. All very standard.

But there are some nice extras you get when you use F# rather than C#. First you can use the double backtick syntax to create more readable names, and second, you can use `let` bound functions in modules rather than classes, which simplifies the code.

```
[<Test>]
let ``When 2 is added to 2 expect 4``() =
    Assert.AreEqual(4, 2+2)
```

The double backtick syntax makes the test results much easier to read. Here is the output of the test with a standard class name:

```
TestClass.When2IsAddedTo2Expect4
Result: Success
```

vs. the output using the more friendly name:

```
MyUnitTests.When 2 is added to 2 expect 4
Result: Success
```

So if you want to write test names that are accessible to non-programmers, give F# a go!

11. Use F# to run unit tests programmatically

Often, you might want to run the unit tests programmatically. This can be for various reasons, such as using custom filters, or doing custom logging, or not wanting to install NUnit on test machines.

One simple way to do this is to use the Fuchu library which lets you organize tests directly, especially parameterized tests, without any complex test attributes.

Here's an example:

```
let add1 x = x + 1

// a simple test using any assertion framework:
// Fuchu's own, Nunit, FsUnit, etc
let ``Assert that add1 is x+1`` x _notUsed =
   NUnit.Framework.Assert.AreEqual(x+1, add1 x)

// a single test case with one value
let simpleTest =
   testCase "Test with 42" <|
     ``Assert that add1 is x+1`` 42

// a parameterized test case with one param
let parameterizedTest i =
   testCase (sprintf "Test with %i" i) <|
     ``Assert that add1 is x+1`` i
```

You can run these tests directly in F# interactive using code like this: `run simpleTest`.

You can also combine these tests into one or more lists, or hierarchical lists of lists:

```
// create a hierarchy of tests
// mark it as the start point with the "Tests" attribute
[<Fuchu.Tests>]
let tests =
   testList "Test group A" [
      simpleTest
      testList "Parameterized 1..10" ([1..10] |> List.map parameterizedTest)
      testList "Parameterized 11..20" ([11..20] |> List.map parameterizedTest)
   ]
```

The code above is available on github.

Finally, with Fuchu, the test assembly becomes its own test runner. Just make the assembly a console app instead of a library and add this code to the `program.fs` file:

```
[<EntryPoint>]
let main args =
    let exitCode = defaultMainThisAssembly args

    Console.WriteLine("Press any key")
    Console.ReadLine() |> ignore

    // return the exit code
    exitCode
```

More on Fuchu here.

Using the NUnit test runner

If you do need to use an existing test runner (such as the NUnit one), then it's very simple to put together a simple script to do this.

I've made a little example, below, using the `Nunit.Runners` package.

All right, this might not be the most exciting use of F#, but it does show off F#'s "object expression" syntax to create the `NUnit.Core.EventListener` interface, so I thought I'd leave it in as a demo.

```
// sets the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

// Requires Nunit.Runners under script directory
//    nuget install NUnit.Runners -o Packages -ExcludeVersion

#r @"Packages\NUnit.Runners\tools\lib\nunit.core.dll"
#r @"Packages\NUnit.Runners\tools\lib\nunit.core.interfaces.dll"

open System
open NUnit.Core

module Setup =
    open System.Reflection
    open NUnit.Core
    open System.Diagnostics.Tracing

    let configureTestRunner path (runner:TestRunner) =
        let package = TestPackage("MyPackage")
        package.Assemblies.Add(path) |> ignore
        runner.Load(package) |> ignore

    let createListener logger =

        let replaceNewline (s:string) =
            s.Replace(Environment.NewLine, "")

        // This is an example of F#'s "object expression" syntax.
        // You don't need to create a class to implement an interface
        {new NUnit.Core.EventListener
            with

            member this.RunStarted(name:string, testCount:int) =
                logger "Run started "

            member this.RunFinished(result:TestResult ) =
                logger ""
                logger "--------------------------------"
                result.ResultState
                |> sprintf "Overall result: %O"
```

```
                |> logger

            member this.RunFinished(ex:Exception) =
                ex.StackTrace
                |> replaceNewline
                |> sprintf "Exception occurred: %s"
                |> logger

            member this.SuiteFinished(result:TestResult) = ()
            member this.SuiteStarted(testName:TestName) = ()

            member this.TestFinished(result:TestResult)=
                result.ResultState
                |> sprintf "Result: %O"
                |> logger

            member this.TestOutput(testOutput:TestOutput) =
                testOutput.Text
                |> replaceNewline
                |> logger

            member this.TestStarted(testName:TestName) =
                logger ""

                testName.FullName
                |> replaceNewline
                |> logger

            member this.UnhandledException(ex:Exception) =
                ex.StackTrace
                |> replaceNewline
                |> sprintf "Unhandled exception occurred: %s"
                |> logger
            }


// run all the tests in the DLL
do
    let dllPath = @".\bin\MyUnitTests.dll"

    CoreExtensions.Host.InitializeService();

    use runner = new NUnit.Core.SimpleTestRunner()
    Setup.configureTestRunner dllPath runner
    let logger = printfn "%s"
    let listener = Setup.createListener logger
    let result = runner.Run(listener, TestFilter.Empty, true, LoggingThreshold.All)

    // if running from the command line, wait for user input
    Console.ReadLine() |> ignore

    // if running from the interactive session, reset session before recompiling MyUni
tTests.dll
```

The code above is available on github.

12. Use F# to learn to write unit tests in other ways

The unit test code above is familiar to all of us, but there are other ways to write tests. Learning to code in different styles is a great way to add some new techniques to your repertoire and expand your thinking in general, so let's have a quick look at some of them.

First up is FsUnit, which replaces `Assert` with a more fluent and idiomatic approach (natural language and piping).

Here's a snippet:

```
open NUnit.Framework
open FsUnit

let inline add x y = x + y

[<Test>]
let ``When 2 is added to 2 expect 4``() =
    add 2 2 |> should equal 4

[<Test>]
let ``When 2.0 is added to 2.0 expect 4.01``() =
    add 2.0 2.0 |> should (equalWithin 0.1) 4.01

[<Test>]
let ``When ToLower(), expect lowercase letters``() =
    "FSHARP".ToLower() |> should startWith "fs"
```

The above code is available on github.

A very different approach is used by Unquote. The Unquote approach is to wrap any F# expression in F# quotations and then evaluate it. If a test expression throws an exception, the test will fail and print not just the exception, but each step up to the point of the exception. This information could potentially give you much more insight in why the assert fails.

Here's a very simple example:

```
[<Test>]
let ``When 2 is added to 2 expect 4``() =
    test <@ 2 + 2 = 4 @>
```

There are also a number of shortcut operators such as `=?` and `>?` that allow you to write your tests even more simply -- no asserts anywhere!

```
[<Test>]
let ``2 + 2 is 4``() =
   let result = 2 + 2
   result =? 4

[<Test>]
let ``2 + 2 is bigger than 5``() =
   let result = 2 + 2
   result >? 5
```

The above code is available on github.

13. Use FsCheck to write better unit tests

The code for this section is available on github.

Let's say that we have written a function that converts numbers to Roman numerals, and we want to create some test cases for it.

We might start writing tests like this:

```
[<Test>]
let ``Test that 497 is CDXCVII``() =
    arabicToRoman 497 |> should equal "CDXCVII"
```

But the problem with this approach is that it only tests a very specific example. There might be some edge cases that we haven't thought of.

A much better approach is to find something that must be true for *all* cases. Then we can create a test that checks that this something (a "property") is true for all cases, or at least a large random subset.

For example, in the Roman numeral example, we can say that one property is "all Roman numerals have at most one 'V' character" or "all Roman numerals have at most three 'X' characters". We can then construct tests that check this property is indeed true.

This is where FsCheck can help. FsCheck is a framework designed for exactly this kind of property-based testing. It's written in F# but it works equally well for testing C# code.

So, let's see how we'd use FsCheck for our Roman numerals.

First, we define some properties that we expect to hold for all Roman numerals.

```
let maxRepetitionProperty ch count (input:string) =
    let find = String.replicate (count+1) ch
    input.Contains find |> not

// a property that holds for all roman numerals
let ``has max rep of one V`` roman =
    maxRepetitionProperty "V" 1 roman

// a property that holds for all roman numerals
let ``has max rep of three Xs`` roman =
    maxRepetitionProperty "X" 3 roman
```

With this in place we create tests that:

1. Create a property checker function suitable for passing to FsCheck.
2. Use the `Check.Quick` function to generate hundreds of random test cases and send them into that property checker.

```
[<Test>]
let ``Test that roman numerals have no more than one V``() =
    let property num =
        // convert the number to roman and check the property
        arabicToRoman num |> ``has max rep of one V``

    Check.QuickThrowOnFailure (testWithRange property)

[<Test>]
let ``Test that roman numerals have no more than three Xs``() =
    let property num =
        // convert the number to roman and check the property
        arabicToRoman num |> ``has max rep of three Xs``

    Check.QuickThrowOnFailure (testWithRange property)
```

Here are the results of the test. You can see that 100 random numbers have been tested, not just one.

```
Test that roman numerals have no more than one V
   Ok, passed 100 tests.

Test that roman numerals have no more than three Xs
   Ok, passed 100 tests.
```

If we changed the test to be "Test that roman numerals have no more than TWO Xs", then the test result is false, and looks like this:

```
Falsifiable, after 33 tests

30
```

In other words, after generating 33 different inputs, FsCheck has correctly found a number (30) that does not meet the required property. Very nice!

Using FsCheck in practice

Not all situations have properties that can be tested this way, but you might find that it is more common than you think.

For example, property based testing is especially useful for "algorithmic" code. Here a few examples:

- If you reverse a list and then reverse it again, you get the original list.
- If you factorize an integer and then multiply the factors, you get the original number.

But even in Boring Line-Of-Business Applications?, you may find that property based testing has a place. For example, here are some things that can be expressed as properties:

- **Roundtripping**. For example, if you save a record to a database and then reload it, the record's fields should be unchanged. Similarly, if you serialize and then deserialize something, you should get the original thing back.
- **Invariants**. If you add products to a sales order, the sum of the individual lines should be the same as the order total. Or, the sum of word counts for each page should be the sum of the word count for the entire book. More generally, if you calculate things via two different paths, you should get the same answer (monoid homomorphisms!)
- **Rounding**. If you add ingredients to a recipe, the sum of the ingredient percentages (with 2 place precision) should always be exactly 100%. Similar rules are needed for most partitioning logic, such as shares, tax calculations, etc. (e.g. the "share pie" example in the DDD book).
  Making sure you get the rounding right in situations like this is where FsCheck shines.

See this SO question for other ideas.

FsCheck is also very useful for doing refactoring, because once you trust that the tests are extremely thorough, you can confidently work on tweaks and optimization.

Some more links for FsCheck:

- I have written an introduction to property-based testing and a follow up on choosing properties for property-based testing.
- FsCheck documentation.

- An article on using FsCheck in practice.
- My post on the Roman Numerals kata that mentions FsCheck.

For more on property-based testing in general, look for articles and videos about QuickCheck.

- Intro to QuickCheck by John Hughes (PDF)
- Fascinating talk on using QuickCheck to find bugs in Riak (another version) (videos)

14. Use FsCheck to create random dummy data

The code for this section is available on github.

In addition to doing testing, FsCheck can be used to create random dummy data.

For example, below is the complete code for generating random customers.

When you combine this with the SQL Type Provider (discussed later) or CSV writer, you can easily generate thousands of rows of random customers in a database or CSV file. Or you can use it with the JSON type provider to call a web service for testing validation logic, or load testing.

(Dont worry about not understanding the code -- this sample is just to show you how easy it is!)

```
// domain objects
type EmailAddress = EmailAddress of string
type PhoneNumber = PhoneNumber of string
type Customer = {
    name: string
    email: EmailAddress
    phone: PhoneNumber
    birthdate: DateTime
    }

// a list of names to sample
let possibleNames = [
    "Georgianne Stephan"
    "Sharolyn Galban"
    "Beatriz Applewhite"
    "Merissa Cornwall"
    "Kenneth Abdulla"
    "Zora Feliz"
    "Janeen Strunk"
    "Oren Curlee"
    ]

// generate a random name by picking from the list at random
```

```
let generateName() =
    FsCheck.Gen.elements possibleNames

// generate a random EmailAddress by combining random users and domains
let generateEmail() =
    let userGen = FsCheck.Gen.elements ["a"; "b"; "c"; "d"; "e"; "f"]
    let domainGen = FsCheck.Gen.elements ["gmail.com"; "example.com"; "outlook.com"]
    let makeEmail u d = sprintf "%s@%s" u d |> EmailAddress
    FsCheck.Gen.map2 makeEmail userGen domainGen

// generate a random PhoneNumber
let generatePhone() =
    let areaGen = FsCheck.Gen.choose(100,999)
    let n1Gen = FsCheck.Gen.choose(1,999)
    let n2Gen = FsCheck.Gen.choose(1,9999)
    let makeNumber area n1 n2 = sprintf "(%03i)%03i-%04i" area n1 n2 |> PhoneNumber
    FsCheck.Gen.map3 makeNumber areaGen n1Gen n2Gen

// generate a random birthdate
let generateDate() =
    let minDate = DateTime(1920,1,1).ToOADate() |> int
    let maxDate = DateTime(2014,1,1).ToOADate() |> int
    let oaDateGen = FsCheck.Gen.choose(minDate,maxDate)
    let makeDate oaDate = float oaDate |> DateTime.FromOADate
    FsCheck.Gen.map makeDate oaDateGen

// a function to create a customer
let createCustomer name email phone birthdate =
    {name=name; email=email; phone=phone; birthdate=birthdate}

// use applicatives to create a customer generator
let generateCustomer =
    createCustomer
    <!> generateName()
    <*> generateEmail()
    <*> generatePhone()
    <*> generateDate()

[<Test>]
let printRandomCustomers() =
    let size = 0
    let count = 10
    let data = FsCheck.Gen.sample size count generateCustomer

    // print it
    data |> List.iter (printfn "%A")
```

And here is a sampling of the results:

```
{name = "Georgianne Stephan";
 email = EmailAddress "d@outlook.com";
 phone = PhoneNumber "(420)330-2080";
 birthdate = 11/02/1976 00:00:00;}

{name = "Sharolyn Galban";
 email = EmailAddress "e@outlook.com";
 phone = PhoneNumber "(579)781-9435";
 birthdate = 01/04/2011 00:00:00;}

{name = "Janeen Strunk";
 email = EmailAddress "b@gmail.com";
 phone = PhoneNumber "(265)405-6619";
 birthdate = 21/07/1955 00:00:00;}
```

15. Use F# to create mocks

If you're using F# to write test cases for code written in C#, you may want to create mocks and stubs for interfaces.

In C# you might use Moq or NSubstitute. In F# you can use object expressions to create interfaces directly, or the Foq library.

Both are easy to do, and in a way that is similar to Moq.

Here's some Moq code in C#:

```
// Moq Method
var mock = new Mock<IFoo>();
mock.Setup(foo => foo.DoSomething("ping")).Returns(true);
var instance = mock.Object;

// Moq Matching Arguments:
mock.Setup(foo => foo.DoSomething(It.IsAny<string>())).Returns(true);

// Moq Property
mock.Setup(foo => foo.Name ).Returns("bar");
```

And here's the equivalent Foq code in F#:

```
// Foq Method
let mock =
    Mock<IFoo>()
        .Setup(fun foo -> <@ foo.DoSomething("ping") @>).Returns(true)
        .Create()

// Foq Matching Arguments
mock.Setup(fun foo -> <@ foo.DoSomething(any()) @>).Returns(true)

// Foq Property
mock.Setup(fun foo -> <@ foo.Name @>).Returns("bar")
```

For more on mocking in F#, see:

- F# as a Unit Testing Language
- Mocking with Foq
- Testing and mocking your C# code with F#

And you need to mock external services such as SMTP over the wire, there is an interesting tool called mountebank, which is easy to interact with in F#.

16. Use F# to do automated browser testing

In addition to unit tests, you should be doing some kind of automated web testing, driving the browser with Selenium or WatiN.

But what language should you write the automation in? Ruby? Python? C#? I think you know the answer!

To make your life even easier, try using Canopy, a web testing framework built on top of Selenium and written in F#. Their site claims *"Quick to learn. Even if you've never done UI Automation, and don't know F#."*, and I'm inclined to believe them.

Below is a snippet taken from the Canopy site. As you can see, the code is simple and easy to understand.

Also, FAKE integrates with Canopy, so you can run automated browser tests as part of a CI build.

```
//start an instance of the firefox browser
start firefox

//this is how you define a test
"taking canopy for a spin" &&& fun _ ->
    //go to url
    url "http://lefthandedgoat.github.io/canopy/testpages/"

    //assert that the element with an id of 'welcome' has
    //the text 'Welcome'
    "#welcome" == "Welcome"

    //assert that the element with an id of 'firstName' has the value 'John'
    "#firstName" == "John"

    //change the value of element with
    //an id of 'firstName' to 'Something Else'
    "#firstName" << "Something Else"

    //verify another element's value, click a button,
    //verify the element is updated
    "#button_clicked" == "button not clicked"
    click "#button"
    "#button_clicked" == "button clicked"

//run all tests
run()
```

17. Use F# for Behaviour Driven Development

The code for this section is available on github.

If you're not familiar with Behaviour Driven Development (BDD), the idea is that you express requirements in a way that is both human-readable and *executable*.

The standard format (Gherkin) for writing these tests uses the Given/When/Then syntax -- here's an example:

```
Feature: Refunded or replaced items should be returned to stock

Scenario 1: Refunded items should be returned to stock
    Given a customer buys a black jumper
    And I have 3 black jumpers left in stock
    When they return the jumper for a refund
    Then I should have 4 black jumpers in stock
```

If you are using BDD already with .NET, you're probably using SpecFlow or similar.

You should consider using TickSpec instead because, as with all things F#, the syntax is much more lightweight.

For example, here's the full implementation of the scenario above.

```
type StockItem = { Count : int }

let mutable stockItem = { Count = 0 }

let [<Given>] ``a customer buys a black jumper`` () =
    ()

let [<Given>] ``I have (.*) black jumpers left in stock`` (n:int) =
    stockItem <- { stockItem with Count = n }

let [<When>] ``they return the jumper for a refund`` () =
    stockItem <- { stockItem with Count = stockItem.Count + 1 }

let [<Then>] ``I should have (.*) black jumpers in stock`` (n:int) =
    let passed = (stockItem.Count = n)
    Assert.True(passed)
```

The C# equivalent has a lot more clutter, and the lack of double backtick syntax really hurts:

```
[Given(@"a customer buys a black jumper")]
public void GivenACustomerBuysABlackJumper()
{
   // code
}

[Given(@"I have (.*) black jumpers left in stock")]
public void GivenIHaveNBlackJumpersLeftInStock(int n)
{
   // code
}
```

Examples taken from the TickSpec site.

Summary of testing in F#

You can of course combine all the test techniques we've seen so far (as this slide deck demonstrates):

- Unit tests (FsUnit, Unquote) and property-based tests (FsCheck).
- Automated acceptance tests (or at least a smoke test) written in BDD (TickSpec) driven by browser automation (Canopy).

- Both types of tests run on every build (with FAKE).

There's a lot of advice on test automation out there, and you'll find that it is easy to port concepts from other languages to these F# tools. Have fun!

Using F# for database related tasks

This post is a continuation of the previous series on low-risk and incremental ways to use F# at work.

In this one, we'll see how F# can be unexpectedly helpful when it comes to database related tasks.

Series contents

Before moving on to the content of the post, here's the full list of the twenty six ways:

Part 1 - Using F# to explore and develop interactively

1. Use F# to explore the .NET framework interactively
2. Use F# to test your own code interactively
3. Use F# to play with webservices interactively
4. Use F# to play with UI's interactively

Part 2 - Using F# for development and devops scripts

5. Use FAKE for build and CI scripts
6. An F# script to check that a website is responding
7. An F# script to convert an RSS feed into CSV
8. An F# script that uses WMI to check the stats of a process
9. Use F# for configuring and managing the cloud

Part 3 - Using F# for testing

10. Use F# to write unit tests with readable names
11. Use F# to run unit tests programmatically
12. Use F# to learn to write unit tests in other ways
13. Use FsCheck to write better unit tests
14. Use FsCheck to create random dummy data
15. Use F# to create mocks
16. Use F# to do automated browser testing
17. Use F# for Behaviour Driven Development

Part 4. Using F# for database related tasks

18. Use F# to replace LINQpad
19. Use F# to unit test stored procedures
20. Use FsCheck to generate random database records
21. Use F# to do simple ETL
22. Use F# to generate SQL Agent scripts

Part 5: Other interesting ways of using F#

23. Use F# for parsing
24. Use F# for diagramming and visualization
25. Use F# for accessing web-based data stores
26. Use F# for data science and machine learning
(BONUS) 27: Balance the generation schedule for the UK power station fleet

---

Part 4. Using F# for database related tasks

This next group of suggestions is all about working with databases, and MS SQL Server in particular.

Relational databases are a critical part of most applications, but most teams do not approach the management of these in the same way as with other development tasks.

For example, how many teams do you know that unit test their stored procedures?

Or their ETL jobs?

Or generate T-SQL admin scripts and other boilerplate using a non-SQL scripting language that's stored in source control?

Here's where F# can shine over other scripting languages, and even over T-SQL itself.

- The database type providers in F# give you the power to create simple, short scripts for testing and admin, with the bonus that...
- The scripts are *type-checked* and will fail at compile time if the database schema changes, which means that...
- The whole process works really well with builds and continuous integration processes, which in turn means that...
- You have really high confidence in your database related code!

We'll look at a few examples to demonstrate what I'm talking about:

- Unit testing stored procedures
- Using FsCheck to generate random records

- Doing simple ETL with F#
- Generating SQL Agent scripts

Getting set up

The code for this section is available on github. In there, there are some SQL scripts to create the sample database, tables and stored procs that I'll use in these examples.

To run the examples, then, you'll need SQL Express or SQL Server running locally or somewhere accessible, with the relevant setup scripts having been run.

Which type provider?

There are a number of SQL Type Providers for F# -- see the fsharp.org Data Access page. For these examples, I'm going to use the `SqlDataConnection` type provider, which is part of the `FSharp.Data.TypeProviders` DLL. It uses SqlMetal behind the scenes and so only works with SQL Server databases.

The SQLProvider project is another good choice -- it supports MySql, SQLite and other non-Microsoft databases.

18. Use F# to replace LINQPad

The code for this section is available on github.

LINQPad is a great tool for doing queries against databases, and is also a general scratchpad for C#/VB/F# code.

You can use F# interactive to do many of the same things -- you get queries, autocompletion, etc., just like LINQPad.

For example, here's one that counts customers with a certain email domain.

```
[<Literal>]
let connectionString = "Data Source=localhost; Initial Catalog=SqlInFsharp; Integrated
 Security=True;"

type Sql = SqlDataConnection<connectionString>
let db = Sql.GetDataContext()

// find the number of customers with a gmail domain
query {
    for c in db.Customer do
    where (c.Email.EndsWith("gmail.com"))
    select c
    count
    }
```

If you want to see what SQL code is generated, you can turn logging on, of course:

```
// optional, turn logging on
db.DataContext.Log <- Console.Out
```

The logged output for this query is:

```
SELECT COUNT(*) AS [value]
FROM [dbo].[Customer] AS [t0]
WHERE [t0].[Email] LIKE @p0
-- @p0: Input VarChar (Size = 8000; Prec = 0; Scale = 0) [%gmail.com]
```

You can also do more complicated things, such as using subqueries. Here's an example from MSDN:

Note that, as befitting a functional approach, queries are nice and composable.

```
// Find students who have signed up at least one course.
query {
    for student in db.Student do
    where (query { for courseSelection in db.CourseSelection do
                   exists (courseSelection.StudentID = student.StudentID) })
    select student
}
```

And if the SQL engine doesn't support certain functions such as regexes, and assuming the size of the data is not too large, you can just stream the data out and do the processing in F#.

```
// find the most popular domain for people born in each decade
let getDomain email =
    Regex.Match(email,".*@(.*)").Groups.[1].Value

let getDecade (birthdate:Nullable<DateTime>) =
    if birthdate.HasValue then
        birthdate.Value.Year / 10  * 10 |> Some
    else
        None

let topDomain list =
    list
    |> Seq.distinct
    |> Seq.head
    |> snd

db.Customer
|> Seq.map (fun c -> getDecade c.Birthdate, getDomain c.Email)
|> Seq.groupBy fst
|> Seq.sortBy fst
|> Seq.map (fun (decade, group) -> (decade,topDomain group))
|> Seq.iter (printfn "%A")
```

As you can see from the code above, the nice thing about doing the processing in F# is that you can define helper functions separately and connect them together easily.

19. Use F# to unit test stored procedures

The code for this section is available on github.

Now let's look at how we can use the type provider to make creating unit tests for stored procs really easy.

First, I create a helper module (which I'll call `DbLib`) to set up the connection and to provide shared utility functions such as `resetDatabase`, which will be called before each test.

```
module DbLib

[<Literal>]
let connectionString = "Data Source=localhost; Initial Catalog=SqlInFsharp;Integrated
Security=True;"
type Sql = SqlDataConnection<connectionString>

let removeExistingData (db:DbContext) =
    let truncateTable name =
        sprintf "TRUNCATE TABLE %s" name
        |> db.DataContext.ExecuteCommand
        |> ignore

    ["Customer"; "CustomerImport"]
    |> List.iter truncateTable

let insertReferenceData (db:DbContext) =
    [ "US","United States";
      "GB","United Kingdom" ]
    |> List.iter (fun (code,name) ->
        let c = new Sql.ServiceTypes.Country()
        c.IsoCode <- code;  c.CountryName <- name
        db.Country.InsertOnSubmit c
        )
    db.DataContext.SubmitChanges()

// removes all data and restores db to known starting point
let resetDatabase() =
    use db = Sql.GetDataContext()
    removeExistingData db
    insertReferenceData db
```

Now I can write a unit test, using NUnit say, just like any other unit test.

Assume that we have `Customer` table, and a sproc called `up_Customer_Upsert` that either inserts a new customer or updates an existing one, depending on whether the passed in customer id is null or not.

Here's what a test looks like:

```
[<Test>]
let ``When upsert customer called with null id, expect customer created with new id``(
) =
    DbLib.resetDatabase()
    use db = DbLib.Sql.GetDataContext()

    // create customer
    let newId = db.Up_Customer_Upsert(Nullable(),"Alice","x@example.com",Nullable())

    // check new id
    Assert.Greater(newId,0)

    // check one customer exists
    let customerCount = db.Customer |> Seq.length
    Assert.AreEqual(1,customerCount)
```

Note that, because the setup is expensive, I do multiple asserts in the test. This could be refactored if you find this too ugly!

Here's one that tests that updates work:

```
[<Test>]
let ``When upsert customer called with existing id, expect customer updated``() =
    DbLib.resetDatabase()
    use db = DbLib.Sql.GetDataContext()

    // create customer
    let custId = db.Up_Customer_Upsert(Nullable(),"Alice","x@example.com",Nullable())

    // update customer
    let newId = db.Up_Customer_Upsert(Nullable custId,"Bob","y@example.com",Nullable()
)

    // check id hasnt changed
    Assert.AreEqual(custId,newId)

    // check still only one customer
    let customerCount = db.Customer |> Seq.length
    Assert.AreEqual(1,customerCount)

    // check customer columns are updated
    let customer = db.Customer |> Seq.head
    Assert.AreEqual("Bob",customer.Name)
```

And one more, that checks for exceptions:

```
[<Test>]
let ``When upsert customer called with blank name, expect validation error``() =
    DbLib.resetDatabase()
    use db = DbLib.Sql.GetDataContext()

    try
        // try to create customer will a blank name
        db.Up_Customer_Upsert(Nullable(),"","x@example.com",Nullable()) |> ignore
        Assert.Fail("expecting a SqlException")
    with
    | :? System.Data.SqlClient.SqlException as ex ->
        Assert.That(ex.Message,Is.StringContaining("@Name"))
        Assert.That(ex.Message,Is.StringContaining("blank"))
```

As you can see, the whole process is very straightforward.

These tests can be compiled and run as part of the continuous integration scripts. And what is great is that, if the database schema gets out of sync with the code, then the tests will fail to even compile!

20. Use FsCheck to generate random database records

The code for this section is available on github.

As I showed in an earlier example, you can use FsCheck to generate random data. In this case we'll use it to generate random records in the database.

Let's say we have a `CustomerImport` table, defined as below. (We'll use this table in the next section on ETL)

```
CREATE TABLE dbo.CustomerImport (
    CustomerId int NOT NULL IDENTITY(1,1)
    ,FirstName varchar(50) NOT NULL
    ,LastName varchar(50) NOT NULL
    ,EmailAddress varchar(50) NOT NULL
    ,Age int NULL

    CONSTRAINT PK_CustomerImport PRIMARY KEY CLUSTERED (CustomerId)
    )
```

Using the same code as before, we can then generate random instances of `CustomerImport`.

```
[<Literal>]
let connectionString = "Data Source=localhost; Initial Catalog=SqlInFsharp; Integrated
 Security=True;"

type Sql = SqlDataConnection<connectionString>

// a list of names to sample
let possibleFirstNames =
    ["Merissa";"Kenneth";"Zora";"Oren"]
let possibleLastNames =
    ["Applewhite";"Feliz";"Abdulla";"Strunk"]

// generate a random name by picking from the list at random
let generateFirstName() =
    FsCheck.Gen.elements possibleFirstNames

let generateLastName() =
    FsCheck.Gen.elements possibleLastNames

// generate a random email address by combining random users and domains
let generateEmail() =
    let userGen = FsCheck.Gen.elements ["a"; "b"; "c"; "d"; "e"; "f"]
    let domainGen = FsCheck.Gen.elements ["gmail.com"; "example.com"; "outlook.com"]
    let makeEmail u d = sprintf "%s@%s" u d
    FsCheck.Gen.map2 makeEmail userGen domainGen
```

So far so good.

Now we get to the `age` column, which is nullable. This means we can't generate random `int` s, but instead we have to generate random `Nullable<int>` s. This is where type checking is really useful -- the compiler has forced us to take that into account. So to make sure we cover all the bases, we'll generate a null value one time out of twenty.

```
// Generate a random nullable age.
// Note that because age is nullable,
// the compiler forces us to take that into account
let generateAge() =
    let nonNullAgeGenerator =
        FsCheck.Gen.choose(1,99)
        |> FsCheck.Gen.map (fun age -> Nullable age)
    let nullAgeGenerator =
        FsCheck.Gen.constant (Nullable())

    // 19 out of 20 times choose a non null age
    FsCheck.Gen.frequency [
        (19,nonNullAgeGenerator)
        (1,nullAgeGenerator)
        ]
```

Putting it altogether...

```
// a function to create a customer
let createCustomerImport first last email age =
    let c = new Sql.ServiceTypes.CustomerImport()
    c.FirstName <- first
    c.LastName <- last
    c.EmailAddress <- email
    c.Age <- age
    c //return new record

// use applicatives to create a customer generator
let generateCustomerImport =
    createCustomerImport
    <!> generateFirstName()
    <*> generateLastName()
    <*> generateEmail()
    <*> generateAge()
```

Once we have a random generator, we can fetch as many records as we like, and insert them using the type provider.

In the code below, we'll generate 10,000 records, hitting the database in batches of 1,000 records.

```
let insertAll() =
    use db = Sql.GetDataContext()

    // optional, turn logging on or off
    // db.DataContext.Log <- Console.Out
    // db.DataContext.Log <- null

    let insertOne counter customer =
        db.CustomerImport.InsertOnSubmit customer
        // do in batches of 1000
        if counter % 1000 = 0 then
            db.DataContext.SubmitChanges()

    // generate the records
    let count = 10000
    let generator = FsCheck.Gen.sample 0 count generateCustomerImport

    // insert the records
    generator |> List.iteri insertOne
    db.DataContext.SubmitChanges() // commit any remaining
```

Finally, let's do it and time it.

```
#time
insertAll()
#time
```

It's not as fast as using BCP, but it is plenty adequate for testing. For example, it only takes a few seconds to create the 10,000 records above.

I want to stress that this is a *single standalone script*, not a heavy binary, so it is really easy to tweak and run on demand.

And of course you get all the goodness of a scripted approach, such as being able to store it in source control, track changes, etc.

21. Use F# to do simple ETL

The code for this section is available on github.

Say that you need to transfer data from one table to another, but it is not a totally straightforward copy, as you need to do some mapping and transformation.

This is a classic ETL (Extract/Transform/Load) situation, and most people will reach for SSIS.

But for some situations, such as one off imports, and where the volumes are not large, you could use F# instead. Let's have a look.

Say that we are importing data into a master table that looks like this:

```
CREATE TABLE dbo.Customer (
    CustomerId int NOT NULL IDENTITY(1,1)
    ,Name varchar(50) NOT NULL
    ,Email varchar(50) NOT NULL
    ,Birthdate datetime NULL
    )
```

But the system we're importing from has a different format, like this:

```
CREATE TABLE dbo.CustomerImport (
    CustomerId int NOT NULL IDENTITY(1,1)
    ,FirstName varchar(50) NOT NULL
    ,LastName varchar(50) NOT NULL
    ,EmailAddress varchar(50) NOT NULL
    ,Age int NULL
    )
```

As part of this import then, we're going to have to:

- Concatenate the `FirstName` and `LastName` columns into one `Name` column
- Map the `EmailAddress` column to the `Email` column
- Calculate a `Birthdate` given an `Age`
- I'm going to skip the `CustomerId` for now -- hopefully we aren't using IDENTITY columns in practice.

The first step is to define a function that maps source records to target records. In this case, we'll call it `makeTargetCustomer`.

Here's some code for this:

```
[<Literal>]
let sourceConnectionString =
    "Data Source=localhost; Initial Catalog=SqlInFsharp; Integrated Security=True;"

[<Literal>]
let targetConnectionString =
    "Data Source=localhost; Initial Catalog=SqlInFsharp; Integrated Security=True;"

type SourceSql = SqlDataConnection<sourceConnectionString>
type TargetSql = SqlDataConnection<targetConnectionString>

let makeName first last =
    sprintf "%s %s" first last

let makeBirthdate (age:Nullable<int>) =
    if age.HasValue then
        Nullable (DateTime.Today.AddYears(-age.Value))
    else
        Nullable()

let makeTargetCustomer (sourceCustomer:SourceSql.ServiceTypes.CustomerImport) =
    let targetCustomer = new TargetSql.ServiceTypes.Customer()
    targetCustomer.Name <- makeName sourceCustomer.FirstName sourceCustomer.LastName
    targetCustomer.Email <- sourceCustomer.EmailAddress
    targetCustomer.Birthdate <- makeBirthdate sourceCustomer.Age
    targetCustomer // return it
```

With this transform in place, the rest of the code is easy, we just just read from the source and write to the target.

```
let transferAll() =
    use sourceDb = SourceSql.GetDataContext()
    use targetDb = TargetSql.GetDataContext()

    let insertOne counter customer =
        targetDb.Customer.InsertOnSubmit customer
        // do in batches of 1000
        if counter % 1000 = 0 then
            targetDb.DataContext.SubmitChanges()
            printfn "...%i records transferred" counter

    // get the sequence of source records
    sourceDb.CustomerImport
    // transform to a target record
    |>  Seq.map makeTargetCustomer
    // and insert
    |>  Seq.iteri insertOne

    targetDb.DataContext.SubmitChanges() // commit any remaining
    printfn "Done"
```

Because these are sequence operations, only one record at a time is in memory (excepting the LINQ submit buffer), so even large data sets can be processed.

To see it in use, first insert a number of records using the dummy data script just discussed, and then run the transfer as follows:

```
#time
transferAll()
#time
```

Again, it only takes a few seconds to transfer 10,000 records.

And again, this is a *single standalone script* -- it's a very lightweight way to create simple ETL jobs.

22. Use F# to generate SQL Agent scripts

For the last database related suggestion, let me suggest the idea of generating SQL Agent scripts from code.

In any decent sized shop you may have hundreds or thousands of SQL Agent jobs. In my opinion, these should all be stored as script files, and loaded into the database when provisioning/building the system.

Alas, there are often subtle differences between dev, test and production environments: connection strings, authorization, alerts, log configuration, etc.

That naturally leads to the problem of trying to keep three different copies of a script around, which in turn makes you think: why not have *one* script and parameterize it for the environment?

But now you are dealing with lots of ugly SQL code! The scripts that create SQL agent jobs are typically hundreds of lines long and were not really designed to be maintained by hand.

F# to the rescue!

In F#, it's really easy to create some simple record types that store all the data you need to generate and configure a job.

For example, in the script below:

- I created a union type called `Step` that could store a `Package`, `Executable`, `Powershell` and so on.
- Each of these step types in turn have their own specific properties, so that a `Package` has a name and variables, and so on.
- A `JobInfo` consists of a name plus a list of `Step` s.
- An agent script is generated from a `JobInfo` plus a set of global properties associated with an environment, such as the databases, shared folder locations, etc.

```
let thisDir = __SOURCE_DIRECTORY__
System.IO.Directory.SetCurrentDirectory (thisDir)

#load @"..\..\SqlAgentLibrary.Lib.fsx"

module MySqlAgentJob =

    open SqlAgentLibrary.Lib.SqlAgentLibrary

    let PackageFolder = @"\shared\etl\MyJob"

    let step1 = Package {
        Name = "An SSIS package"
        Package = "AnSsisPackage.dtsx"
        Variables =
            [
            "EtlServer", "EtlServer"
            "EtlDatabase", "EtlDatabase"
            "SsisLogServer", "SsisLogServer"
            "SsisLogDatabase", "SsisLogDatabase"
            ]
        }

    let step2 = Package {
```

```
        Name = "Another SSIS package"
        Package = "AnotherSsisPackage.dtsx"
        Variables =
            [
            "EtlServer", "EtlServer2"
            "EtlDatabase", "EtlDatabase2"
            "SsisLogServer", "SsisLogServer2"
            "SsisLogDatabase", "SsisLogDatabase2"
            ]
        }

    let jobInfo = {
        JobName = "My SqlAgent Job"
        JobDescription = "Copy data from one place to another"
        JobCategory = "ETL"
        Steps =
            [
            step1
            step2
            ]
        StepsThatContinueOnFailure = []
        JobSchedule = None
        JobAlert = None
        JobNotification = None
        }

    let generate globals =
        writeAgentScript globals jobInfo

module DevEnvironment =

    let globals =
        [
        // global
        "Environment", "DEV"
        "PackageFolder", @"\shared\etl\MyJob"
        "JobServer", "(local)"

        // General variables
        "JobName", "Some packages"
        "SetStartFlag", "2"
        "SetEndFlag", "0"

        // databases
        "Database", "mydatabase"
        "Server",  "localhost"
        "EtlServer", "localhost"
        "EtlDatabase", "etl_config"

        "SsisLogServer", "localhost"
        "SsisLogDatabase", "etl_config"
        ] |> Map.ofList
```

```
    let generateJob() =
        MySqlAgentJob.generate globals

DevEnvironment.generateJob()
```

I can't share the actual F# code, but I think you get the idea. It's quite simple to create.

Once we have these .FSX files, we can generate the real SQL Agent scripts en-masse and then deploy them to the appropriate servers.

Below is an example of a SQL Agent script that might be generated automatically from the .FSX file.

As you can see, it is a nicely laid out and formatted T-SQL script. The idea is that a DBA can review it and be confident that no magic is happening, and thus be willing to accept it as input.

On the other hand, it would be risky to maintain scripts like. Editing the SQL code directly could be risky. Better to use type-checked (and more concise) F# code than untyped T-SQL!

```
USE [msdb]
GO

-- =====================================================
-- Script that deletes and recreates the SQL Agent job 'My SqlAgent Job'
--
-- The job steps are:
-- 1) An SSIS package
     -- {Continue on error=false}
-- 2) Another SSIS package
     -- {Continue on error=false}

-- =====================================================



-- =====================================================
-- Environment is DEV
--
-- The other global variables are:
-- Database = mydatabase
-- EtlDatabase = etl_config
-- EtlServer = localhost
-- JobName = My SqlAgent Job
-- JobServer = (local)
-- PackageFolder = \\shared\etl\MyJob\
-- Server = localhost
-- SetEndFlag = 0
-- SetStartFlag = 2
-- SsisLogDatabase = etl_config
```

```
-- SsisLogServer = localhost

-- =====================================================


-- =====================================================
-- Create job
-- =====================================================

-- ---------------------------------------------
-- Delete Job if it exists
-- ---------------------------------------------
IF  EXISTS (SELECT job_id FROM msdb.dbo.sysjobs_view WHERE name = 'My SqlAgent Job')
BEGIN
    PRINT 'Deleting job "My SqlAgent Job"'
    EXEC msdb.dbo.sp_delete_job @job_name='My SqlAgent Job', @delete_unused_schedule=0
END

-- ---------------------------------------------
-- Create Job
-- ---------------------------------------------

BEGIN TRANSACTION
DECLARE @ReturnCode INT
SELECT @ReturnCode = 0

-- ---------------------------------------------
-- Create Category if needed
-- ---------------------------------------------
IF NOT EXISTS (SELECT name FROM msdb.dbo.syscategories WHERE name='ETL' AND category_c
lass=1)
BEGIN
    PRINT 'Creating category "ETL"'
    EXEC @ReturnCode = msdb.dbo.sp_add_category @class=N'JOB', @type=N'LOCAL', @name='
ETL'
    IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback
END

-- ---------------------------------------------
-- Create Job
-- ---------------------------------------------

DECLARE @jobId BINARY(16)
PRINT 'Creating job "My SqlAgent Job"'
EXEC @ReturnCode =  msdb.dbo.sp_add_job @job_name='My SqlAgent Job',
        @enabled=1,
        @category_name='ETL',
        @owner_login_name=N'sa',
        @description='Copy data from one place to another',
        @job_id = @jobId OUTPUT

IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback
```

```
PRINT '-- -----------------------------------------------'
PRINT 'Create step 1: "An SSIS package"'
PRINT '-- -----------------------------------------------'
DECLARE @Step1_Name nvarchar(50) = 'An SSIS package'
DECLARE @Step1_Package nvarchar(170) = 'AnSsisPackage.dtsx'
DECLARE @Step1_Command nvarchar(1700) =
    '/FILE "\\shared\etl\MyJob\AnSsisPackage.dtsx"' +
    ' /CHECKPOINTING OFF' +
    ' /SET "\Package.Variables[User::SetFlag].Value";"2"' +
    ' /SET "\Package.Variables[User::JobName].Value";""' +
    ' /SET "\Package.Variables[User::SourceServer].Value";"localhost"' +
    ' /SET "\Package.Variables[User::SourceDatabaseName].Value";"etl_config"' +

    ' /REPORTING E'

EXEC @ReturnCode = msdb.dbo.sp_add_jobstep @job_id=@jobId, @step_name=@Step1_Name,
        @step_id=1,
        @on_success_action=3,
        @on_fail_action=2,
        @subsystem=N'SSIS',
        @command=@Step1_Command

        IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback


PRINT '-- -----------------------------------------------'
PRINT 'Create step 2: "Another SSIS Package"'
PRINT '-- -----------------------------------------------'
DECLARE @Step2_Name nvarchar(50) = 'Another SSIS Package'
DECLARE @Step2_Package nvarchar(170) = 'AnotherSsisPackage.dtsx'
DECLARE @Step2_Command nvarchar(1700) =
    '/FILE "\\shared\etl\MyJob\AnotherSsisPackage.dtsx.dtsx"' +
    ' /CHECKPOINTING OFF' +
    ' /SET "\Package.Variables[User::EtlServer].Value";"localhost"' +
    ' /SET "\Package.Variables[User::EtlDatabase].Value";"etl_config"' +
    ' /SET "\Package.Variables[User::SsisLogServer].Value";"localhost"' +
    ' /SET "\Package.Variables[User::SsisLogDatabase].Value";"etl_config"' +

    ' /REPORTING E'

EXEC @ReturnCode = msdb.dbo.sp_add_jobstep @job_id=@jobId, @step_name=@Step2_Name,
        @step_id=2,
        @on_success_action=3,
        @on_fail_action=2,
        @subsystem=N'SSIS',
        @command=@Step2_Command

        IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

    -- -----------------------------------------------
-- Job Schedule
-- -----------------------------------------------
```

```
-- --------------------------------------------------
-- Job Alert
-- --------------------------------------------------



-- -------------------------------------------------
-- Set start step
-- -------------------------------------------------

EXEC @ReturnCode = msdb.dbo.sp_update_job @job_id = @jobId, @start_step_id = 1
IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback

-- -------------------------------------------------
-- Set server
-- -------------------------------------------------


EXEC @ReturnCode = msdb.dbo.sp_add_jobserver @job_id = @jobId, @server_name = '(local)'

IF (@@ERROR <> 0 OR @ReturnCode <> 0) GOTO QuitWithRollback


PRINT 'Done!'

COMMIT TRANSACTION
GOTO EndSave
QuitWithRollback:
    IF (@@TRANCOUNT > 0) ROLLBACK TRANSACTION
EndSave:
GO
```

Summary

I hope that this set of suggestions has thrown a new light on what F# can be used for.

In my opinion, the combination of concise syntax, lightweight scripting (no binaries) and SQL type providers makes F# incredibly useful for database related tasks.

Please leave a comment and let me know what you think.

Other interesting ways of using F# at work

This post is the conclusion of the series on low-risk and incremental ways to use F# at work.

To wrap up, we'll look at a few more ways in which F# can help you with various development tasks around the edges, without impacting any core or mission critical code.

Series contents

Before moving on to the content of the post, here's the full list of the twenty six ways:

Part 1 - Using F# to explore and develop interactively

1. Use F# to explore the .NET framework interactively
2. Use F# to test your own code interactively
3. Use F# to play with webservices interactively
4. Use F# to play with UI's interactively

Part 2 - Using F# for development and devops scripts

5. Use FAKE for build and CI scripts
6. An F# script to check that a website is responding
7. An F# script to convert an RSS feed into CSV
8. An F# script that uses WMI to check the stats of a process
9. Use F# for configuring and managing the cloud

Part 3 - Using F# for testing

10. Use F# to write unit tests with readable names
11. Use F# to run unit tests programmatically
12. Use F# to learn to write unit tests in other ways
13. Use FsCheck to write better unit tests
14. Use FsCheck to create random dummy data
15. Use F# to create mocks
16. Use F# to do automated browser testing
17. Use F# for Behaviour Driven Development

Part 4. Using F# for database related tasks

18. Use F# to replace LINQpad
19. Use F# to unit test stored procedures
20. Use FsCheck to generate random database records

21. Use F# to do simple ETL
22. Use F# to generate SQL Agent scripts

Part 5: Other interesting ways of using F#

23. Use F# for parsing
24. Use F# for diagramming and visualization
25. Use F# for accessing web-based data stores
26. Use F# for data science and machine learning
(BONUS) 27: Balance the generation schedule for the UK power station fleet

---

Part 5: Other ways of using F# outside the core

This last group of suggestions is a bit of a mish-mash I'm afraid. These are things that didn't fit into earlier posts, mostly concerning using F# for analysis and data processing.

23. Use F# for parsing

It is surprising how often you need to parse something in the course of routine development: splitting strings at spaces, reading a CSV file, doing substitutions in a template, finding HTML links for a web crawler, parsing a query string in a URI, and so on.

F#, being an ML-derived language, is ideal for parsing tasks of all kinds, from simple regexes to full fledged parsers.

Of course, there are many off-the-shelf libraries for common tasks, but sometimes you need to write your own. A good example of this is TickSpec, the BDD framework that we saw earlier.

TickSpec needs to parse the so-called "Gherkin" format of Given/When/Then. Rather than create a dependency on another library, I imagine that it was easier (and more fun) for Phil to write his own parser in a few hundred lines. You can see part of the source code here.

Another situation where it might be worth writing your own parser is when you have some complex system, such as a rules engine, which has a horrible XML configuration format. Rather than manually editing the configuration, you could create a very simple domain specific language (DSL) that is parsed and then converted to the complex XML.

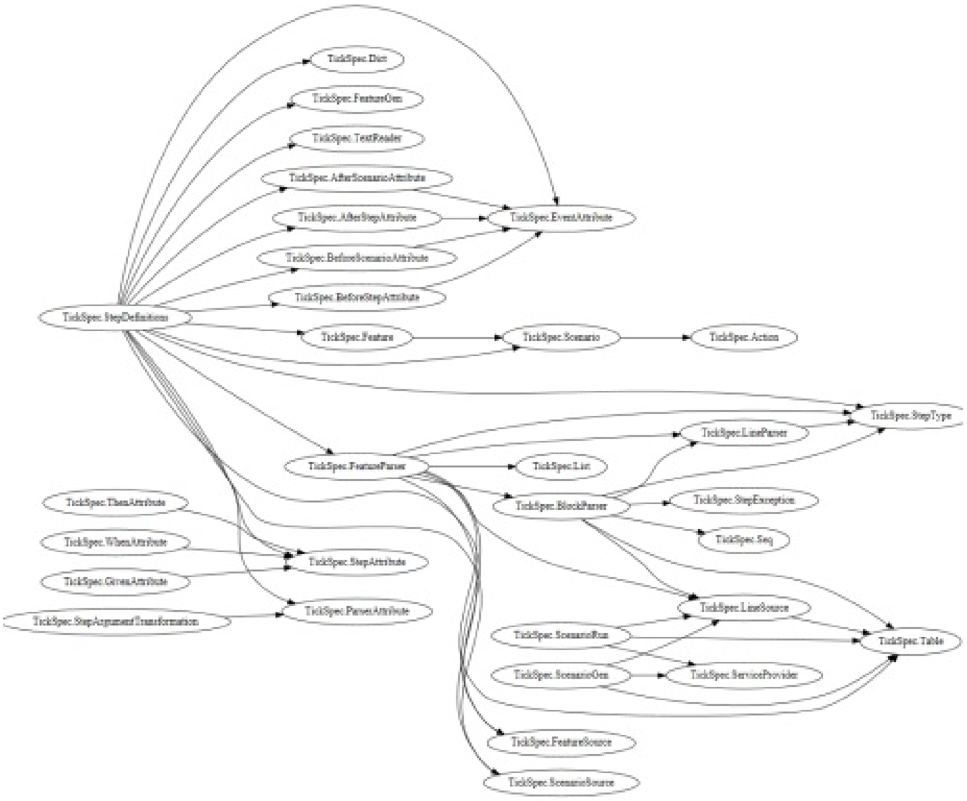
In his book on DSLs, Martin Fowler gives an example of this, a DSL that is parsed to create a state machine. And here is an F# implementation of that DSL.

For more complicating parsing tasks, I highly recommend using FParsec, which is perfectly suited for this kind of thing. For example, it has been used for parsing search queries for FogCreek, CSV files, chess notation, and a custom DSL for load testing scenarios.

24. Use F# for diagramming and visualization

Once you have parsed or analyzed something, it is always nice if you can display the results visually, rather than as tables full of data.

For example, in a previous post I used F# in conjunction with GraphViz to create diagrams of dependency relationships. You can see a sample below:



The code to generate the diagram itself was short, only about 60 lines, which you can see here.

As an alternative to GraphViz, you could also consider using FSGraph.

For more mathematical or data-centric visualizations, there are a number of good libraries:

- FSharp.Charting for desktop visualizations that is well integrated with F# scripting.
- FsPlot for interactive visualizations in HTML.
- VegaHub, an F# library for working with Vega
- F# for Visualization

And finally, there's the 800 lb gorilla -- Excel.

Using the built-in capabilities of Excel is great, if it is available. And F# scripting plays well with Excel.

You can chart in Excel, plot functions in Excel, and for even more power and integration, you have the FCell and Excel-DNA projects.

25. Use F# for accessing web-based data stores

There is a lot of public data out on the web, just waiting to pulled down and loved. With the magic of type providers, F# is a good choice for direct integrating these web-scale data stores into your workflow.

Right now, we'll look at two data stores: Freebase and World Bank. More will be available soon -- see the fsharp.org Data Access page for the latest information.

Freebase

The code for this section is available on github.

Freebase is a large collaborative knowledge base and online collection of structured data harvested from many sources.

To get started, just link in the type provider DLL as we have seen before.

The site is throttled, so you'll probably need an API key if you're using it a lot (api details here)

```
// sets the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

// Requires FSharp.Data under script directory
//    nuget install FSharp.Data -o Packages -ExcludeVersion
#r @"Packages\FSharp.Data\lib\net40\FSharp.Data.dll"
open FSharp.Data

// without a key
let data = FreebaseData.GetDataContext()

// with a key
(*
[<Literal>]
let FreebaseApiKey = "<enter your freebase-enabled google API key here>"
type FreebaseDataWithKey = FreebaseDataProvider<Key=FreebaseApiKey>
let data = FreebaseDataWithKey.GetDataContext()
*)
```

Once the type provider is loaded, you can start asking questions, such as...

"Who are the US presidents?"

```
data.Society.Government.``US Presidents``
|> Seq.map (fun p ->  p.``President number`` |> Seq.head, p.Name)
|> Seq.sortBy fst
|> Seq.iter (fun (n,name) -> printfn "%s was number %i" name n )
```

Result:

```
George Washington was number 1
John Adams was number 2
Thomas Jefferson was number 3
James Madison was number 4
James Monroe was number 5
John Quincy Adams was number 6
...
Ronald Reagan was number 40
George H. W. Bush was number 41
Bill Clinton was number 42
George W. Bush was number 43
Barack Obama was number 44
```

Not bad for just four lines of code!

How about *"what awards did Casablanca win?"*

```
data.``Arts and Entertainment``.Film.Films.IndividualsAZ.C.Casablanca.``Awards Won``
|> Seq.map (fun award -> award.Year, award.``Award category``.Name)
|> Seq.sortBy fst
|> Seq.iter (fun (year,name) -> printfn "%s -- %s" year name)
```

The result is:

```
1943 -- Academy Award for Best Director
1943 -- Academy Award for Best Picture
1943 -- Academy Award for Best Screenplay
```

So that's Freebase. Lots of good information, both useful and frivolous.

More on how to use the Freebase type provider.

Using Freebase to generate realistic test data

We've seen how FsCheck can be used to generate test data. Well, you can also get the same affect by getting data from Freebase, which makes the data much more realistic.

Kit Eason showed how to do this in a tweet, and here's an example based on his code:

```
let randomElement =
    let random = new System.Random()
    fun (arr:string array) -> arr.[random.Next(arr.Length)]

let surnames =
    FreebaseData.GetDataContext().Society.People.``Family names``
    |> Seq.truncate 1000
    |> Seq.map (fun name -> name.Name)
    |> Array.ofSeq

let firstnames =
    FreebaseData.GetDataContext().Society.Celebrities.Celebrities
    |> Seq.truncate 1000
    |> Seq.map (fun celeb -> celeb.Name.Split([|' '|]).[0])
    |> Array.ofSeq

// generate ten random people and print
type Person = {Forename:string; Surname:string}
Seq.init 10 ( fun _ ->
    {Forename = (randomElement firstnames);
     Surname = (randomElement surnames) }
     )
|> Seq.iter (printfn "%A")
```

The results are:

```
{Forename = "Kelly"; Surname = "Deasy";}
{Forename = "Bam"; Surname = "Br?z?";}
{Forename = "Claire"; Surname = "Sludden";}
{Forename = "Kenneth"; Surname = "Kl?tz";}
{Forename = "?tienne"; Surname = "Defendi";}
{Forename = "Billy"; Surname = "Paleti";}
{Forename = "Alix"; Surname = "Nuin";}
{Forename = "Katherine"; Surname = "Desporte";}
{Forename = "Jasmine";  Surname = "Belousov";}
{Forename = "Josh";  Surname = "Kramarsic";}
```

World Bank

The code for this section is available on github.

On the other extreme from Freebase is the World Bank Open Data, which has lots of detailed economic and social information from around the world.

The setup is identical to Freebase, but no API key is needed.

```
// sets the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

// Requires FSharp.Data under script directory
//    nuget install FSharp.Data -o Packages -ExcludeVersion
#r @"Packages\FSharp.Data\lib\net40\FSharp.Data.dll"
open FSharp.Data

let data = WorldBankData.GetDataContext()
```

With the type provider set up, we can do a serious query, such as:

"How do malnutrition rates compare between low income and high income countries?"

```
// Create a list of countries to process
let groups =
 [| data.Countries.``Low income``
    data.Countries.``High income``
    |]

// get data from an indicator for particular year
let getYearValue (year:int) (ind:Runtime.WorldBank.Indicator) =
    ind.Name,year,ind.Item year

// get data
[ for c in groups ->
    c.Name,
    c.Indicators.``Malnutrition prevalence, weight for age (% of children under 5)`` |
> getYearValue 2010
]
// print the data
|> Seq.iter (
    fun (group,(indName, indYear, indValue)) ->
       printfn "%s -- %s %i %0.2f%% " group indName indYear indValue)
```

The result is:

```
Low income -- Malnutrition prevalence, weight for age (% of children under 5) 2010 23.
19%
High income -- Malnutrition prevalence, weight for age (% of children under 5) 2010 1.
36%
```

Similarly, here is the code to compare maternal mortality rates:

```
// Create a list of countries to process
let countries =
 [| data.Countries.``European Union``
    data.Countries.``United Kingdom``
    data.Countries.``United States`` |]

/ get data
[ for c in countries  ->
    c.Name,
    c.Indicators.``Maternal mortality ratio (modeled estimate, per 100,000 live births
)`` |> getYearValue 2010
]
// print the data
|> Seq.iter (
    fun (group,(indName, indYear, indValue)) ->
       printfn "%s -- %s %i %0.1f" group indName indYear indValue)
```

The result is:

```
European Union -- Maternal mortality ratio (modeled estimate, per 100,000 live births)
 2010 9.0
United Kingdom -- Maternal mortality ratio (modeled estimate, per 100,000 live births)
 2010 12.0
United States -- Maternal mortality ratio (modeled estimate, per 100,000 live births)
2010 21.0
```

More on how to use the World Bank type provider.

26. Use F# for data science and machine learning

So you're putting all these suggestions into practice. You're parsing your web logs with FParsec, extracting stats from your internal databases with the SQL type provider, and pulling down external data from web services. You've got all this data -- what can you do with it?

Let's finish up by having a quick look at using F# for data science and machine learning.

As we have seen, F# is great for exploratory programming -- it has a REPL with intellisense. But unlike Python and R, your code is type checked, so you know that your code is not going to fail with an exception halfway through a two hour processing job!

If you are familiar with the Pandas library from Python or the ?tseries' package in R, then you should take a serious look at Deedle, an easy-to-use, high quality package for data and time series manipulation. Deedle is designed to work well for exploratory programming using the REPL, but can be also used in efficient compiled .NET code.

And if you use R a lot, there's an R type provider(of course). This means you can use R packages as if they were .NET libraries. How awesome is that!

There's lots of other F# friendly packages too. You can find out all about them at fsharp.org.

- Data science
- Math
- Machine learning

Series summary

Phew! That was a long list of examples and a lot of code to look at. If you've made it to the end, congratulations!

I hope that this has given you some new insights into the value of F#. It's not just a math-y or financial language -- it's a practical one too. And it can help you with all sorts of things in your development, testing, and data management workflows.

Finally, as I have stressed throughout this series, all these uses are safe, low risk and incremental. What's the worst that can happen?

So go on, persuade your team mates and boss to give F# a try, and let me know how it goes.

Postscript

After I posted this, Simon Cousins tweeted that I missed one -- I can't resist adding it.

> @ScottWlaschin 27: balance the generation schedule for the uk power station fleet. seriously, the alternative to #fsharp was way too risky
>
> — Simon Cousins (@simontcousins) April 25, 2014

You can read more about Simon's real-world of use of F# (for power generation) on his blog. There are more testimonials to F# at fsharp.org.

This series of posts will give you a guided tour through the main features of F# and then show you ways that F# can help you in your day-to-day development.

- Introduction to the 'Why use F#' series. An overview of the benefits of F#.
- F# syntax in 60 seconds. A very quick overview on how to read F# code.
- Comparing F# with C#: A simple sum. In which we attempt to sum the squares from 1 to N without using a loop.
- Comparing F# with C#: Sorting. In which we see that F# is more declarative than C#, and we are introduced to pattern matching..
- Comparing F# with C#: Downloading a web page. In which we see that F# excels at callbacks, and we are introduced to the 'use' keyword.
- Four Key Concepts. The concepts that differentiate F# from a standard imperative language.
- Conciseness. Why is conciseness important?.
- Type inference. How to avoid getting distracted by complex type syntax.
- Low overhead type definitions. No penalty for making new types.
- Using functions to extract boilerplate code. The functional approach to the DRY principle.
- Using functions as building blocks. Function composition and mini-languages make code more readable.
- Pattern matching for conciseness. Pattern matching can match and bind in a single step.
- Convenience. Features that reduce programming drudgery and boilerplate code.
- Out-of-the-box behavior for types. Immutability and built-in equality with no coding.
- Functions as interfaces. OO design patterns can be trivial when functions are used.
- Partial Application. How to fix some of a function's parameters.
- Active patterns. Dynamic patterns for powerful matching.
- Correctness. How to write 'compile time unit tests'.
- Immutability. Making your code predictable.
- Exhaustive pattern matching. A powerful technique to ensure correctness.
- Using the type system to ensure correct code. In F# the type system is your friend, not your enemy.
- Worked example: Designing for correctness. How to make illegal states unrepresentable.
- Concurrency. The next major revolution in how we write software?.
- Asynchronous programming. Encapsulating a background task with the Async class.
- Messages and Agents. Making it easier to think about concurrency.
- Functional Reactive Programming. Turning events into streams.
- Completeness. F# is part of the whole .NET ecosystem.
- Seamless interoperation with .NET libraries. Some convenient features for working with .NET libraries.

- Anything C# can do.... A whirlwind tour of object-oriented code in F#.
- Why use F#: Conclusion. .

Introduction to the 'Why use F#' series

This series of posts will give you a guided tour through the main features of F# and then show you ways that F# can help you in your day-to-day development.

Key benefits of F# compared with C#

If you are already familiar with C# or Java, you might be wondering why it would be worth learning yet another language. F# has some major benefits which I have grouped under the following themes:

- **Conciseness**. F# is not cluttered up with coding "noise" such as curly brackets, semicolons and so on. You almost never have to specify the type of an object, thanks to a powerful type inference system. And it generally takes less lines of code to solve the same problem.
- **Convenience**. Many common programming tasks are much simpler in F#. This includes things like creating and using complex type definitions, doing list processing, comparison and equality, state machines, and much more. And because functions are first class objects, it is very easy to create powerful and reusable code by creating functions that have other functions as parameters, or that combine existing functions to create new functionality.
- **Correctness**. F# has a very powerful type system which prevents many common errors such as null reference exceptions. And in addition, you can often encode business logic using the type system itself, so that it is actually impossible to write incorrect code, because it is caught at compile time as a type error.
- **Concurrency**. F# has a number of built-in tools and libraries to help with programming systems when more than one thing at a time is happening. Asynchronous programming is directly supported, as is parallelism. F# also has a message queuing system, and excellent support for event handling and reactive programming. And because data structures are immutable by default, sharing state and avoiding locks is much easier.
- **Completeness**. Although F# is a functional language at heart, it does support other styles which are not 100% pure, which makes it much easier to interact with the non-pure world of web sites, databases, other applications, and so on. In particular, F# is designed as a hybrid functional/OO language, so it can do almost everything that C# can do as well. Of course, F# integrates seamlessly with the .NET ecosystem, which gives you access to all the third party .NET libraries and tools. Finally, it is part of Visual Studio, which means you get a good editor with IntelliSense support, a debugger, and many plug-ins for unit tests, source control, and other development tasks.

In the rest of this series of posts, I will try to demonstrate each of these F# benefits, using standalone snippets of F# code (and often with C# code for comparison). I'll briefly cover all the major features of F#, including pattern matching, function composition, and concurrent programming. By the time you have finished this series, I hope that you will have been impressed with the power and elegance of F#, and you will be encouraged to use it for your next project!

How to read and use the example code

All the code snippets in these posts have been designed to be run interactively. I strongly recommend that you evaluate the snippets as you read each post. The source for any large code files will be linked to from the post.

This series is not a tutorial, so I will not go too much into *why* the code works. Don't worry if you cannot understand some of the details; the goal of the series is just to introduce you to F# and whet your appetitite for learning it more deeply.

If you have experience in languages such as C# and Java, you have probably found that you can get a pretty good understanding of source code written in other similar languages, even if you aren't familiar with the keywords or the libraries. You might ask "how do I assign a variable?" or "how do I do a loop?", and with these answers be able to do some basic programming quite quickly.

This approach will not work for F#, because in its pure form there are no variables, no loops, and no objects. Don't be frustrated - it will eventually make sense! If you want to learn F# in more depth, there are some helpful tips on the "learning F#" page.

Comparing F# with C#: A simple sum

To see what some real F# code looks like, let's start with a simple problem: "sum the squares from 1 to N".

We'll compare an F# implementation with a C# implementation. First, the F# code:

```
// define the square function
let square x = x * x

// define the sumOfSquares function
let sumOfSquares n =
   [1..n] |> List.map square |> List.sum

// try it
sumOfSquares 100
```

The mysterious looking `|>` is called the pipe operator. It just pipes the output of one expression into the input of the next. So the code for `sumOfSquares` reads as:

1. Create a list of 1 to n (square brackets construct a list).
2. Pipe the list into the library function called `List.map`, transforming the input list into an output list using the "square" function we just defined.
3. Pipe the resulting list of squares into the library function called `List.sum`. Can you guess what it does?
4. There is no explicit "return" statement. The output of `List.sum` is the overall result of the function.

Next, here's a C# implementation using the classic (non-functional) style of a C-based language. (A more functional version using LINQ is discussed later.)

```
public static class SumOfSquaresHelper
{
   public static int Square(int i)
   {
      return i * i;
   }

   public static int SumOfSquares(int n)
   {
      int sum = 0;
      for (int i = 1; i <= n; i++)
      {
         sum += Square(i);
      }
      return sum;
   }
}
```

What are the differences?

- The F# code is more compact
- The F# code didn't have any type declarations
- F# can be developed interactively

Let's take each of these in turn.

Less code

The most obvious difference is that there is a lot more C# code. 13 C# lines compared with 3 F# lines (ignoring comments). The C# code has lots of "noise", things like curly braces, semicolons, etc. And in C# the functions cannot stand alone, but need to be added to some class ("SumOfSquaresHelper"). F# uses whitespace instead of parentheses, needs no line terminator, and the functions can stand alone.

In F# it is common for entire functions to be written on one line, as the "square" function is. The `sumOfSquares` function could also have been written on one line. In C# this is normally frowned upon as bad practice.

When a function does have multiple lines, F# uses indentation to indicate a block of code, which eliminates the need for braces. (If you have ever used Python, this is the same idea). So the `sumOfSquares` function could also have been written this way:

```
let sumOfSquares n =
   [1..n]
   |> List.map square
   |> List.sum
```

The only drawback is that you have to indent your code carefully. Personally, I think it is worth the trade-off.

No type declarations

The next difference is that the C# code has to explicitly declare all the types used. For example, the `int i` parameter and `int SumOfSquares` return type. Yes, C# does allow you to use the "var" keyword in many places, but not for parameters and return types of functions.

In the F# code we didn't declare any types at all. This is an important point: F# looks like an untyped language, but it is actually just as type-safe as C#, in fact, even more so! F# uses a technique called "type inference" to infer the types you are using from their context. It works amazingly very well most of the time, and reduces the code complexity immensely.

In this case, the type inference algorithm notes that we started with a list of integers. That in turn implies that the square function and the sum function must be taking ints as well, and that the final value must be an int. You can see what the inferred types are by looking at the result of the compilation in the interactive window. You'll see something like:

```
val square : int -> int
```

which means that the "square" function takes an int and returns an int.

If the original list had used floats instead, the type inference system would have deduced that the square function used floats instead. Try it and see:

```
// define the square function
let squareF x = x * x

// define the sumOfSquares function
let sumOfSquaresF n =
   [1.0 .. n] |> List.map squareF |> List.sum  // "1.0" is a float

sumOfSquaresF 100.0
```

The type checking is very strict! If you try using a list of floats (`[1.0..n]`) in the original `sumOfSquares` example, or a list of ints (`[1 ..n]`) in the `sumOfSquaresF` example, you will get a type error from the compiler.

Interactive development

Finally, F# has an interactive window where you can test the code immediately and play around with it. In C# there is no easy way to do this.

For example, I can write my square function and immediately test it:

```
// define the square function
let square x = x * x

// test
let s2 = square 2
let s3 = square 3
let s4 = square 4
```

When I am satisfied that it works, I can move on to the next bit of code.

This kind of interactivity encourages an incremental approach to coding that can become addictive!

Furthermore, many people claim that designing code interactively enforces good design practices such as decoupling and explicit dependencies, and therefore, code that is suitable for interactive evaluation will also be code that is easy to test. Conversely, code that cannot be tested interactively will probably be hard to test as well.

The C# code revisited

My original example was written using "old-style" C#. C# has incorporated a lot of functional features, and it is possible to rewrite the example in a more compact way using the LINQ extensions.

So here is another C# version -- a line-for-line translation of the F# code.

```
public static class FunctionalSumOfSquaresHelper
{
   public static int SumOfSquares(int n)
   {
      return Enumerable.Range(1, n)
         .Select(i => i * i)
         .Sum();
   }
}
```

However, in addition to the noise of the curly braces and periods and semicolons, the C# version needs to declare the parameter and return types, unlike the F# version.

Many C# developers may find this a trivial example, but still resort back to loops when the logic becomes more complicated. In F# though, you will almost never see explicit loops like this. See for example, this post on eliminating boilerplate from more complicated loops.

Comparing F# with C#: Sorting

In this next example, we will implement a quicksort-like algorithm for sorting lists and compare an F# implementation to a C# implementation.

Here is the logic for a simplified quicksort-like algorithm:

```
If the list is empty, there is nothing to do.
Otherwise:
  1. Take the first element of the list
  2. Find all elements in the rest of the list that
      are less than the first element, and sort them.
  3. Find all elements in the rest of the list that
      are >= than the first element, and sort them
  4. Combine the three parts together to get the final result:
      (sorted smaller elements + firstElement +
       sorted larger elements)
```

Note that this is a simplified algorithm and is not optimized (and it does not sort in place, like a true quicksort); we want to focus on clarity for now.

Here is the code in F#:

```
let rec quicksort list =
   match list with
   | [] ->                           // If the list is empty
        []                           // return an empty list
   | firstElem::otherElements ->     // If the list is not empty
        let smallerElements =        // extract the smaller ones
            otherElements
            |> List.filter (fun e -> e < firstElem)
            |> quicksort             // and sort them
        let largerElements =         // extract the large ones
            otherElements
            |> List.filter (fun e -> e >= firstElem)
            |> quicksort             // and sort them
        // Combine the 3 parts into a new list and return it
        List.concat [smallerElements; [firstElem]; largerElements]

//test
printfn "%A" (quicksort [1;5;23;18;9;1;3])
```

Again note that this is not an optimized implementation, but is designed to mirror the algorithm closely.

Let's go through this code:

- There are no type declarations anywhere. This function will work on any list that has comparable items (which is almost all F# types, because they automatically have a default comparison function).
- The whole function is recursive -- this is signaled to the compiler using the `rec` keyword in " `let rec quicksort list =` ".
- The `match..with` is sort of like a switch/case statement. Each branch to test is signaled with a vertical bar, like so:

```
match x with
| caseA -> something
| caseB -> somethingElse
```

- The " `match` " with `[]` matches an empty list, and returns an empty list.
- The " `match` " with `firstElem::otherElements` does two things.
  - First, it only matches a non-empty list.
  - Second, it creates two new values automatically. One for the first element called " `firstElem` ", and one for the rest of the list, called " `otherElements` ". In C# terms, this is like having a "switch" statement that not only branches, but does variable declaration and assignment *at the same time*.
- The `->` is sort of like a lambda (`=>`) in C#. The equivalent C# lambda would look something like `(firstElem, otherElements) => do something` .
- The " `smallerElements` " section takes the rest of the list, filters it against the first element using an inline lambda expression with the " `<` " operator and then pipes the result into the quicksort function recursively.
- The " `largerElements` " line does the same thing, except using the " `>=` " operator
- Finally the resulting list is constructed using the list concatenation function " `List.concat` ". For this to work, the first element needs to be put into a list, which is what the square brackets are for.
- Again note there is no "return" keyword; the last value will be returned. In the " `[]` " branch the return value is the empty list, and in the main branch, it is the newly constructed list.

For comparison here is an old-style C# implementation (without using LINQ).

```
public class QuickSortHelper
{
   public static List<T> QuickSort<T>(List<T> values)
      where T : IComparable
   {
      if (values.Count == 0)
      {
         return new List<T>();
      }

      //get the first element
      T firstElement = values[0];

      //get the smaller and larger elements
      var smallerElements = new List<T>();
      var largerElements = new List<T>();
      for (int i = 1; i < values.Count; i++)  // i starts at 1
      {                                       // not 0!
         var elem = values[i];
         if (elem.CompareTo(firstElement) < 0)
         {
            smallerElements.Add(elem);
         }
         else
         {
            largerElements.Add(elem);
         }
      }

      //return the result
      var result = new List<T>();
      result.AddRange(QuickSort(smallerElements.ToList()));
      result.Add(firstElement);
      result.AddRange(QuickSort(largerElements.ToList()));
      return result;
   }
}
```

Comparing the two sets of code, again we can see that the F# code is much more compact, with less noise and no need for type declarations.

Furthermore, the F# code reads almost exactly like the actual algorithm, unlike the C# code. This is another key advantage of F# -- the code is generally more declarative ("what to do") and less imperative ("how to do it") than C#, and is therefore much more self-documenting.

A functional implementation in C#

Here's a more modern "functional-style" implementation using LINQ and an extension method:

```
public static class QuickSortExtension
{
    /// <summary>
    /// Implement as an extension method for IEnumerable
    /// </summary>
    public static IEnumerable<T> QuickSort<T>(
        this IEnumerable<T> values) where T : IComparable
    {
        if (values == null || !values.Any())
        {
            return new List<T>();
        }

        //split the list into the first element and the rest
        var firstElement = values.First();
        var rest = values.Skip(1);

        //get the smaller and larger elements
        var smallerElements = rest
                .Where(i => i.CompareTo(firstElement) < 0)
                .QuickSort();

        var largerElements = rest
                .Where(i => i.CompareTo(firstElement) >= 0)
                .QuickSort();

        //return the result
        return smallerElements
            .Concat(new List<T>{firstElement})
            .Concat(largerElements);
    }
}
```

This is much cleaner, and reads almost the same as the F# version. But unfortunately there is no way of avoiding the extra noise in the function signature.

Correctness

Finally, a beneficial side-effect of this compactness is that F# code often works the first time, while the C# code may require more debugging.

Indeed, when coding these samples, the old-style C# code was incorrect initially, and required some debugging to get it right. Particularly tricky areas were the `for` loop (starting at 1 not zero) and the `CompareTo` comparison (which I got the wrong way round), and it

would also be very easy to accidentally modify the inbound list. The functional style in the second C# example is not only cleaner but was easier to code correctly.

But even the functional C# version has drawbacks compared to the F# version. For example, because F# uses pattern matching, it is not possible to branch to the "non-empty list" case with an empty list. On the other hand, in the C# code, if we forgot the test:

```
if (values == null || !values.Any()) ...
```

then the extraction of the first element:

```
var firstElement = values.First();
```

would fail with an exception. The compiler cannot enforce this for you. In your own code, how often have you used `FirstOrDefault` rather than `First` because you are writing "defensive" code. Here is an example of a code pattern that is very common in C# but is rare in F#:

```
var item = values.FirstOrDefault();  // instead of .First()
if (item != null)
{
   // do something if item is valid
}
```

The one-step "pattern match and branch" in F# allows you to avoid this in many cases.

Postscript

The example implementation in F# above is actually pretty verbose by F# standards!

For fun, here is what a more typically concise version would look like:

```
let rec quicksort2 = function
   | [] -> []
   | first::rest ->
        let smaller,larger = List.partition ((>=) first) rest
        List.concat [quicksort2 smaller; [first]; quicksort2 larger]

// test code
printfn "%A" (quicksort2 [1;5;23;18;9;1;3])
```

Not bad for 4 lines of code, and when you get used to the syntax, still quite readable.

Comparing F# with C#: Downloading a web page

In this example, we will compare the F# and C# code for downloading a web page, with a callback to process the text stream.

We'll start with a straightforward F# implementation.

```
// "open" brings a .NET namespace into visibility
open System.Net
open System
open System.IO

// Fetch the contents of a web page
let fetchUrl callback url =
    let req = WebRequest.Create(Uri(url))
    use resp = req.GetResponse()
    use stream = resp.GetResponseStream()
    use reader = new IO.StreamReader(stream)
    callback reader url
```

Let's go through this code:

- The use of "open" at the top allows us to write "WebRequest" rather than "System.Net.WebRequest". It is similar to a " `using System.Net` " header in C#.
- Next, we define the `fetchUrl` function, which takes two arguments, a callback to process the stream, and the url to fetch.
- We next wrap the url string in a Uri. F# has strict type-checking, so if instead we had written: `let req = WebRequest.Create(url)` the compiler would have complained that it didn't know which version of `WebRequest.Create` to use.
- When declaring the `response` , `stream` and `reader` values, the " `use` " keyword is used instead of " `let` ". This can only be used in conjunction with classes that implement `IDisposable` . It tells the compiler to automatically dispose of the resource when it goes out of scope. This is equivalent to the C# " `using` " keyword.
- The last line calls the callback function with the StreamReader and url as parameters. Note that the type of the callback does not have to be specified anywhere.

Now here is the equivalent C# implementation.

```
class WebPageDownloader
{
    public TResult FetchUrl<TResult>(
        string url,
        Func<string, StreamReader, TResult> callback)
    {
        var req = WebRequest.Create(url);
        using (var resp = req.GetResponse())
        {
            using (var stream = resp.GetResponseStream())
            {
                using (var reader = new StreamReader(stream))
                {
                    return callback(url, reader);
                }
            }
        }
    }
}
```

As usual, the C# version has more 'noise'.

- There are ten lines just for curly braces, and there is the visual complexity of 5 levels of nesting*
- All the parameter types have to be explicitly declared, and the generic `TResult` type has to be repeated three times.

* It's true that in this particular example, when all the `using` statements are adjacent, the extra braces and indenting can be removed, but in the more general case they are needed.

Testing the code

Back in F# land, we can now test the code interactively:

```
let myCallback (reader:IO.StreamReader) url =
    let html = reader.ReadToEnd()
    let html1000 = html.Substring(0,1000)
    printfn "Downloaded %s. First 1000 is %s" url html1000
    html      // return all the html

//test
let google = fetchUrl myCallback "http://google.com"
```

Finally, we have to resort to a type declaration for the reader parameter (`reader:IO.StreamReader`). This is required because the F# compiler cannot determine the type of the "reader" parameter automatically.

A very useful feature of F# is that you can "bake in" parameters in a function so that they don't have to be passed in every time. This is why the `url` parameter was placed *last* rather than first, as in the C# version. The callback can be setup once, while the url varies from call to call.

```
// build a function with the callback "baked in"
let fetchUrl2 = fetchUrl myCallback

// test
let google = fetchUrl2 "http://www.google.com"
let bbc    = fetchUrl2 "http://news.bbc.co.uk"

// test with a list of sites
let sites = ["http://www.bing.com";
             "http://www.google.com";
             "http://www.yahoo.com"]

// process each site in the list
sites |> List.map fetchUrl2
```

The last line (using `List.map`) shows how the new function can be easily used in conjunction with list processing functions to download a whole list at once.

Here is the equivalent C# test code:

```
[Test]
public void TestFetchUrlWithCallback()
{
    Func<string, StreamReader, string> myCallback = (url, reader) =>
    {
        var html = reader.ReadToEnd();
        var html1000 = html.Substring(0, 1000);
        Console.WriteLine(
            "Downloaded {0}. First 1000 is {1}", url,
            html1000);
        return html;
    };

    var downloader = new WebPageDownloader();
    var google = downloader.FetchUrl("http://www.google.com",
                                      myCallback);

    // test with a list of sites
    var sites = new List<string> {
        "http://www.bing.com",
        "http://www.google.com",
        "http://www.yahoo.com"};

    // process each site in the list
    sites.ForEach(site => downloader.FetchUrl(site, myCallback));
}
```

Again, the code is a bit noisier than the F# code, with many explicit type references. More importantly, the C# code doesn't easily allow you to bake in some of the parameters in a function, so the callback must be explicitly referenced every time.

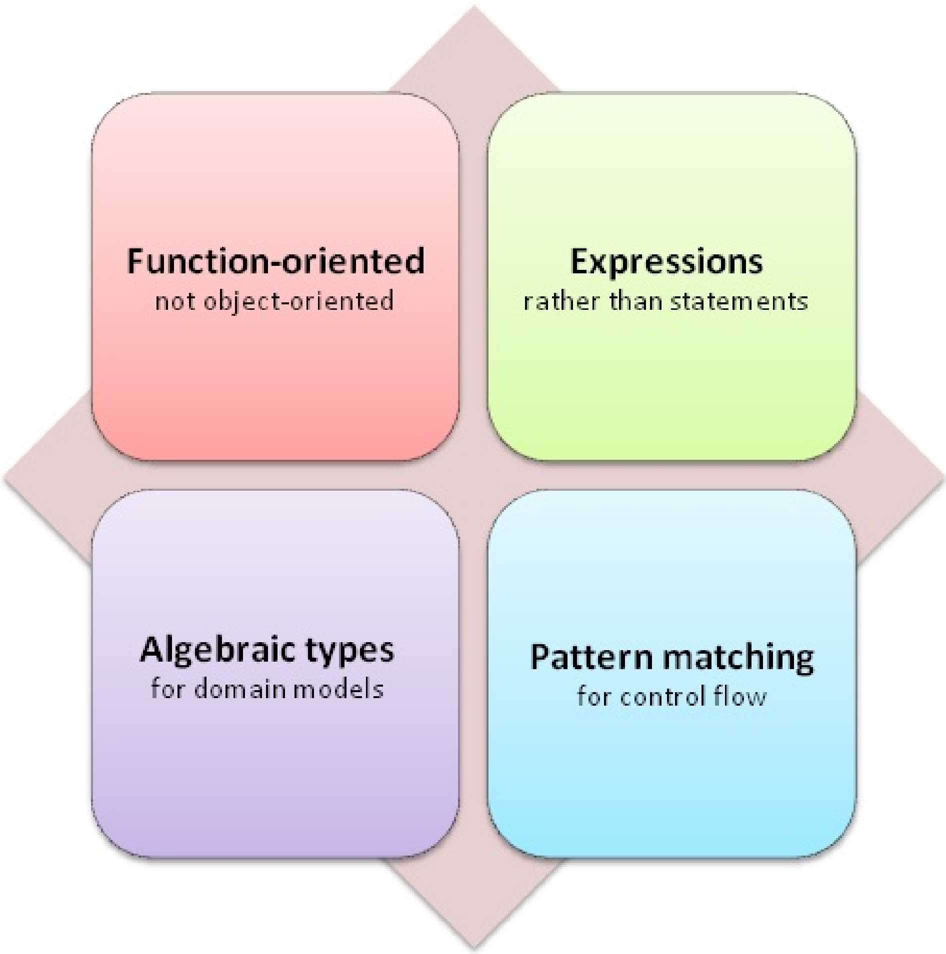
Four Key Concepts

In the next few posts we'll move on to demonstrating the themes of this series: conciseness, convenience, correctness, concurrency and completeness.

But before that, let's look at some of the key concepts in F# that we will meet over and over again. F# is different in many ways from a standard imperative language like C#, but there are a few major differences that are particularly important to understand:

- **Function-oriented** rather than object-oriented
- **Expressions** rather than statements
- **Algebraic types** for creating domain models
- **Pattern matching** for flow of control

In later posts, these will be dealt with in much greater depth -- this is just a taster to help you understand the rest of this series.



Function-oriented rather than object-oriented

As you might expect from the term "functional programming", functions are everywhere in F#.

Of course, functions are first class entities, and can be passed around like any other value:

```
let square x = x * x

// functions as values
let squareclone = square
let result = [1..10] |> List.map squareclone

// functions taking other functions as parameters
let execFunction aFunc aParam = aFunc aParam
let result2 = execFunction square 12
```

But C# has first-class functions too, so what's so special about functional programming?

The short answer is that the function-oriented nature of F# infiltrates every part of the language and type system in a way that it does not in C#, so that things that are awkward or clumsy in C# are very elegant in F#.

It's hard to explain this in a few paragraphs, but here are some of the benefits that we will see demonstrated over this series of posts:

- **Building with composition**. Composition is the 'glue' that allows us build larger systems from smaller ones. This is not an optional technique, but is at the very heart of the functional style. Almost every line of code is a composable expression (see below). Composition is used to build basic functions, and then functions that use those functions, and so on. And the composition principle doesn't just apply to functions, but also to types (the product and sum types discussed below).
- **Factoring and refactoring**. The ability to factor a problem into parts depends how easily the parts can be glued back together. Methods and classes that might seem to be indivisible in an imperative language can often be broken down into surprisingly small pieces in a functional design. These fine-grained components typically consist of (a) a few very general functions that take other functions as parameters, and (b) other helper functions that specialize the general case for a particular data structure or application. Once factored out, the generalized functions allow many additional operations to be programmed very easily without having to write new code. You can see a good example of a general function like this (the fold function) in the post on extracting duplicate code from loops.
- **Good design**. Many of the principles of good design, such as "separation of concerns", "single responsibility principle", "program to an interface, not an implementation", arise naturally as a result of a functional approach. And functional code tends to be high level and declarative in general.

The following posts in this series will have examples of how functions can make code more concise and convenient, and then for a deeper understanding, there is a whole series on thinking functionally.

Expressions rather than statements

In functional languages, there are no statements, only expressions. That is, every chunk of code always returns a value, and larger chunks are created by combining smaller chunks using composition rather than a serialized list of statements.

If you have used LINQ or SQL you will already be familiar with expression-based languages. For example, in pure SQL, you cannot have assignments. Instead, you must have subqueries within larger queries.

```
SELECT EmployeeName
FROM Employees
WHERE EmployeeID IN
    (SELECT DISTINCT ManagerID FROM Employees)  -- subquery
```

F# works in the same way -- every function definition is a single expression, not a set of statements.

And it might not be obvious, but code built from expressions is both safer and more compact than using statements. To see this, let's compare some statement-based code in C# with the equivalent expression-based code.

First, the statement-based code. Statements don't return values, so you have to use temporary variables that are assigned to from within statement bodies.

```
// statement-based code in C#
int result;
if (aBool)
{
  result = 42;
}
Console.WriteLine("result={0}", result);
```

Because the `if-then` block is a statement, the `result` variable must be defined *outside* the statement but assigned to from *inside* the statement, which leads to issues such as:

- What initial value should `result` be set to?
- What if I forget to assign to the `result` variable?
- What is the value of the `result` variable in the "else" case?

For comparison, here is the same code, rewritten in an expression-oriented style:

```
// expression-based code in C#
int result = (aBool) ? 42 : 0;
Console.WriteLine("result={0}", result);
```

In the expression-oriented version, none of these issues apply:

- The `result` variable is declared at the same time that it is assigned. No variables have to be set up "outside" the expression and there is no worry about what initial value they should be set to.
- The "else" is explicitly handled. There is no chance of forgetting to do an assignment in one of the branches.
- It is not possible to forget to assign `result`, because then the variable would not even exist!

Expression-oriented style is not a choice in F#, and it is one of the things that requires a change of approach when coming from an imperative background.

Algebraic Types

The type system in F# is based on the concept of **algebraic types**. That is, new compound types are built by combining existing types in two different ways:

- First, a combination of values, each picked from a set of types. These are called "product" types.
- Of, alternately, as a disjoint union representing a choice between a set of types. These are called "sum" types.

For example, given existing types `int` and `bool`, we can create a new product type that must have one of each:

```
//declare it
type IntAndBool = {intPart: int; boolPart: bool}

//use it
let x = {intPart=1; boolPart=false}
```

Alternatively, we can create a new union/sum type that has a choice between each type:

```
//declare it
type IntOrBool =
    | IntChoice of int
    | BoolChoice of bool

//use it
let y = IntChoice 42
let z = BoolChoice true
```

These "choice" types are not available in C#, but are incredibly useful for modeling many real-world cases, such as states in a state machine (which is a surprisingly common theme in many domains).

And by combining "product" and "sum" types in this way, it is easy to create a rich set of types that accurately models any business domain. For examples of this in action, see the posts on low overhead type definitions and using the type system to ensure correct code.

Pattern matching for flow of control

Most imperative languages offer a variety of control flow statements for branching and looping:

- `if-then-else` (and the ternary version `bool ? if-true : if-false`)
- `case` or `switch` statements
- `for` and `foreach` loops, with `break` and `continue`
- `while` and `until` loops
- and even the dreaded `goto`

F# does support some of these, but F# also supports the most general form of conditional expression, which is **pattern-matching**.

A typical matching expression that replaces `if-then-else` looks like this:

```
match booleanExpression with
| true -> // true branch
| false -> // false branch
```

And the replacement of `switch` might look like this:

```
match aDigit with
| 1 -> // Case when digit=1
| 2 -> // Case when digit=2
| _ -> // Case otherwise
```

Finally, loops are generally done using recursion, and typically look something like this:

```
match aList with
| [] ->
     // Empty case
| first::rest ->
     // Case with at least one element.
     // Process first element, and then call
     // recursively with the rest of the list
```

Although the match expression seems unnecessarily complicated at first, you'll see that in practice it is both elegant and powerful.

For the benefits of pattern matching, see the post on exhaustive pattern matching, and for a worked example that uses pattern matching heavily, see the roman numerals example.

Pattern matching with union types

We mentioned above that F# supports a "union" or "choice" type. This is used instead of inheritance to work with different variants of an underlying type. Pattern matching works seamlessly with these types to create a flow of control for each choice.

In the following example, we create a `Shape` type representing four different shapes and then define a `draw` function with different behavior for each kind of shape. This is similar to polymorphism in an object oriented language, but based on functions.

```
type Shape =        // define a "union" of alternative structures
| Circle of int
| Rectangle of int * int
| Polygon of (int * int) list
| Point of (int * int)

let draw shape =    // define a function "draw" with a shape param
  match shape with
  | Circle radius ->
      printfn "The circle has a radius of %d" radius
  | Rectangle (height,width) ->
      printfn "The rectangle is %d high by %d wide" height width
  | Polygon points ->
      printfn "The polygon is made of these points %A" points
  | _ -> printfn "I don't recognize this shape"

let circle = Circle(10)
let rect = Rectangle(4,5)
let polygon = Polygon( [(1,1); (2,2); (3,3)])
let point = Point(2,3)

[circle; rect; polygon; point] |> List.iter draw
```

A few things to note:

- As usual, we didn't have to specify any types. The compiler correctly determined that the shape parameter for the "draw" function was of type `Shape`.
- You can see that the `match..with` logic not only matches against the internal structure of the shape, but also assigns values based on what is appropriate for the shape.
- The underscore is similar to the "default" branch in a switch statement, except that in F# it is required -- every possible case must always be handled. If you comment out the line

  ```
  | _ -> printfn "I don't recognize this shape"
  ```

see what happens when you compile!

These kinds of choice types can be simulated somewhat in C# by using subclasses or interfaces, but there is no built in support in the C# type system for this kind of exhaustive matching with error checking.

Conciseness

After having seen some simple code, we will now move on to demonstrating the major themes (conciseness, convenience, correctness, concurrency and completeness), filtered through the concepts of types, functions and pattern matching.

With the next few posts, we'll examine the features of F# that aid conciseness and readability.

An important goal for most mainstream programming languages is a good balance of readability and conciseness. Too much conciseness can result in hard-to-understand or obfuscated code (APL anyone?), while too much verbosity can easily swamp the underlying meaning. Ideally, we want a high signal-to-noise ratio, where every word and character in the code contributes to the meaning of the code, and there is minimal boilerplate.

Why is conciseness important? Here are a few reasons:

- **A concise language tends to be more declarative**, saying *what* the code should do rather than *how* to do it. That is, declarative code is more focused on the high-level logic rather than the nuts and bolts of the implementation.
- **It is easier to reason about correctness** if there are fewer lines of code to reason about!
- And of course, **you can see more code on a screen** at a time. This might seem trivial, but the more you can see, the more you can grasp as well.

As you have seen, compared with C#, F# is generally much more concise. This is due to features such as:

- **Type inference** and **low overhead type definitions**. One of the major reasons for F#'s conciseness and readability is its type system. F# makes it very easy to create new types as you need them. They don't cause visual clutter either in their definition or in use, and the type inference system means that you can use them freely without getting distracted by complex type syntax.
- **Using functions to extract boilerplate code**. The DRY principle ("don't repeat yourself") is a core principle of good design in functional languages as well as object-oriented languages. In F# it is extremely easy to extract repetitive code into common utility functions, which allows you to focus on the important stuff.
- **Composing complex code from simple functions** and **creating mini-languages**. The functional approach makes it easy to create a set of basic operations and then combine these building blocks in various ways to build up more complex behaviors. In this way, even the most complex code is still very concise and readable.

- **Pattern matching**. We've seen pattern matching as a glorified switch statement, but in fact it is much more general, as it can compare expressions in a number of ways, matching on values, conditions, and types, and then assign or extract values, all at the same time.

Type inference

As you have already seen, F# uses a technique called "type inference" to greatly reduce the number of type annotations that need to be explicitly specified in normal code. And even when types do need to be specified, the syntax is less longwinded compared to C#.

To see this, here are some C# methods that wrap two standard LINQ functions. The implementations are trivial, but the method signatures are extremely complex:

```
public IEnumerable<TSource> Where<TSource>(
    IEnumerable<TSource> source,
    Func<TSource, bool> predicate
    )
{
    //use the standard LINQ implementation
    return source.Where(predicate);
}

public IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
    )
{
    //use the standard LINQ implementation
    return source.GroupBy(keySelector);
}
```

And here are the exact F# equivalents, showing that no type annotations are needed at all!

```
let Where source predicate =
    //use the standard F# implementation
    Seq.filter predicate source

let GroupBy source keySelector =
    //use the standard F# implementation
    Seq.groupBy keySelector source
```

You might notice that the standard F# implementations for "filter" and "groupBy" have the parameters in exactly the opposite order from the LINQ implementations used in C#. The "source" parameter is placed last, rather than first. There is a reason for this, which will be explained in the thinking functionally series.

The type inference algorithm is excellent at gathering information from many sources to determine the types. In the following example, it correctly deduces that the `list` value is a list of strings.

```
let i  = 1
let s = "hello"
let tuple  = s,i      // pack into tuple
let s2,i2  = tuple    // unpack
let list = [s2]       // type is string list
```

And in this example, it correctly deduces that the `sumLengths` function takes a list of strings and returns an int.

```
let sumLengths strList =
    strList |> List.map String.length |> List.sum

// function type is: string list -> int
```

Low overhead type definitions

In C#, there is a disincentive for creating new types ? the lack of type inference means you need to explicitly specify types in most places, resulting in brittleness and more visual clutter. As a result, there is always a temptation to create monolithic classes rather than modularizing them.

In F# there is no penalty for making new types, so it is quite common to have hundreds if not thousands of them. Every time you need to define a structure, you can create a special type, rather than reusing (and overloading) existing types such as strings and lists.

This means that your programs will be more type-safe, more self documenting, and more maintainable (because when the types change you will immediately get compile-time errors rather than runtime errors).

Here are some examples of one-liner types in F#:

```
open System

// some "record" types
type Person = {FirstName:string; LastName:string; Dob:DateTime}
type Coord = {Lat:float; Long:float}

// some "union" (choice) types
type TimePeriod = Hour | Day | Week | Year
type Temperature = C of int | F of int
type Appointment = OneTime of DateTime
                   | Recurring of DateTime list
```

F# types and domain driven design

The conciseness of the type system in F# is particularly useful when doing domain driven design (DDD). In DDD, for each real world entity and value object, you ideally want to have a corresponding type. This can mean creating hundreds of "little" types, which can be tedious in C#.

Furthermore, "value" objects in DDD should have structural equality, meaning that two objects containing the same data should always be equal. In C# this can mean more tedium in overriding `IEquatable<T>` , but in F#, you get this for free by default.

To show how easy it is to create DDD types in F#, here are some example types that might be created for a simple "customer" domain.

```
type PersonalName = {FirstName:string; LastName:string}

// Addresses
type StreetAddress = {Line1:string; Line2:string; Line3:string }

type ZipCode =  ZipCode of string
type StateAbbrev =  StateAbbrev of string
type ZipAndState =  {State:StateAbbrev; Zip:ZipCode }
type USAddress = {Street:StreetAddress; Region:ZipAndState}

type UKPostCode =  PostCode of string
type UKAddress = {Street:StreetAddress; Region:UKPostCode}

type InternationalAddress = {
    Street:StreetAddress; Region:string; CountryName:string}

// choice type  -- must be one of these three specific types
type Address = USAddress | UKAddress | InternationalAddress

// Email
type Email = Email of string

// Phone
type CountryPrefix = Prefix of int
type Phone = {CountryPrefix:CountryPrefix; LocalNumber:string}

type Contact =
    {
    PersonalName: PersonalName;
    // "option" means it might be missing
    Address: Address option;
    Email: Email option;
    Phone: Phone option;
    }

// Put it all together into a CustomerAccount type
type CustomerAccountId  = AccountId of string
type CustomerType  = Prospect | Active | Inactive

// override equality and deny comparison
[<CustomEquality; NoComparison>]
type CustomerAccount =
    {
    CustomerAccountId: CustomerAccountId;
    CustomerType: CustomerType;
    ContactInfo: Contact;
    }

    override this.Equals(other) =
        match other with
        | :? CustomerAccount as otherCust ->
          (this.CustomerAccountId = otherCust.CustomerAccountId)
```

```
        | _ -> false

    override this.GetHashCode() = hash this.CustomerAccountId
```

This code fragment contains 17 type definitions in just a few lines, but with minimal complexity. How many lines of C# code would you need to do the same thing?

Obviously, this is a simplified version with just the basic types ? in a real system, constraints and other methods would be added. But note how easy it is to create lots of DDD value objects, especially wrapper types for strings, such as "`ZipCode`" and "`Email`". By using these wrapper types, we can enforce certain constraints at creation time, and also ensure that these types don't get confused with unconstrained strings in normal code. The only "entity" type is the `CustomerAccount`, which is clearly indicated as having special treatment for equality and comparison.

For a more in-depth discussion, see the series called "Domain driven design in F#".

Using functions to extract boilerplate code

In the very first example in this series, we saw a simple function that calculated the sum of squares, implemented in both F# and C#. Now let's say we want some new functions which are similar, such as:

- Calculating the product of all the numbers up to N
- Counting the sum of odd numbers up to N
- The alternating sum of the numbers up to N

Obviously, all these requirements are similar, but how would you extract any common functionality?

Let's start with some straightforward implementations in C# first:

```
public static int Product(int n)
{
    int product = 1;
    for (int i = 1; i <= n; i++)
    {
        product *= i;
    }
    return product;
}

public static int SumOfOdds(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; i++)
    {
        if (i % 2 != 0) { sum += i; }
    }
    return sum;
}

public static int AlternatingSum(int n)
{
    int sum = 0;
    bool isNeg = true;
    for (int i = 1; i <= n; i++)
    {
        if (isNeg)
        {
            sum -= i;
            isNeg = false;
        }
        else
        {
            sum += i;
            isNeg = true;
        }
    }
    return sum;
}
```

What do all these implementations have in common? The looping logic! As programmers, we are told to remember the DRY principle ("don't repeat yourself"), yet here we have repeated almost exactly the same loop logic each time. Let's see if we can extract just the differences between these three methods:

Function	Initial value	Inner loop logic
Product	product=1	Multiply the i'th value with the running total
SumOfOdds	sum=0	Add the i'th value to the running total if not even
AlternatingSum	int sum = 0 bool isNeg = true	Use the isNeg flag to decide whether to add or subtract, and flip the flag for the next pass.

Is there a way to strip the duplicate code and focus on the just the setup and inner loop logic? Yes there is. Here are the same three functions in F#:

```
let product n =
    let initialValue = 1
    let action productSoFar x = productSoFar * x
    [1..n] |> List.fold action initialValue

//test
product 10

let sumOfOdds n =
    let initialValue = 0
    let action sumSoFar x = if x%2=0 then sumSoFar else sumSoFar+x
    [1..n] |> List.fold action initialValue

//test
sumOfOdds 10

let alternatingSum n =
    let initialValue = (true,0)
    let action (isNeg,sumSoFar) x = if isNeg then (false,sumSoFar-x)
                                            else (true ,sumSoFar+x)
    [1..n] |> List.fold action initialValue |> snd

//test
alternatingSum 100
```

All three of these functions have the same pattern:

1. Set up the initial value
2. Set up an action function that will be performed on each element inside the loop.
3. Call the library function `List.fold`. This is a powerful, general purpose function which starts with the initial value and then runs the action function for each element in the list in turn.

The action function always has two parameters: a running total (or state) and the list element to act on (called "x" in the above examples).

In the last function, `alternatingSum`, you will notice that it used a tuple (pair of values) for the initial value and the result of the action. This is because both the running total and the `isNeg` flag must be passed to the next iteration of the loop -- there are no "global" values that can be used. The final result of the fold is also a tuple so we have to use the "snd" (second) function to extract the final total that we want.

By using `List.fold` and avoiding any loop logic at all, the F# code gains a number of benefits:

- **The key program logic is emphasized and made explicit**. The important differences between the functions become very clear, while the commonalities are pushed to the background.
- **The boilerplate loop code has been eliminated**, and as a result the code is more condensed than the C# version (4-5 lines of F# code vs. at least 9 lines of C# code)
- **There can never be a error in the loop logic** (such as off-by-one) because that logic is not exposed to us.

By the way, the sum of squares example could also be written using `fold` as well:

```
let sumOfSquaresWithFold n =
    let initialValue = 0
    let action sumSoFar x = sumSoFar + (x*x)
    [1..n] |> List.fold action initialValue

//test
sumOfSquaresWithFold 100
```

"Fold" in C#

Can you use the "fold" approach in C#? Yes. LINQ does have an equivalent to `fold`, called `Aggregate`. And here is the C# code rewritten to use it:

```
public static int ProductWithAggregate(int n)
{
    var initialValue = 1;
    Func<int, int, int> action = (productSoFar, x) =>
        productSoFar * x;
    return Enumerable.Range(1, n)
            .Aggregate(initialValue, action);
}

public static int SumOfOddsWithAggregate(int n)
{
    var initialValue = 0;
    Func<int, int, int> action = (sumSoFar, x) =>
        (x % 2 == 0) ? sumSoFar : sumSoFar + x;
    return Enumerable.Range(1, n)
        .Aggregate(initialValue, action);
}

public static int AlternatingSumsWithAggregate(int n)
{
    var initialValue = Tuple.Create(true, 0);
    Func<Tuple<bool, int>, int, Tuple<bool, int>> action =
        (t, x) => t.Item1
            ? Tuple.Create(false, t.Item2 - x)
            : Tuple.Create(true, t.Item2 + x);
    return Enumerable.Range(1, n)
        .Aggregate(initialValue, action)
        .Item2;
}
```

Well, in some sense these implementations are simpler and safer than the original C# versions, but all the extra noise from the generic types makes this approach much less elegant than the equivalent code in F#. You can see why most C# programmers prefer to stick with explicit loops.

A more relevant example

A slightly more relevant example that crops up frequently in the real world is how to get the "maximum" element of a list when the elements are classes or structs. The LINQ method 'max' only returns the maximum value, not the whole element that contains the maximum value.

Here's a solution using an explicit loop:

```
public class NameAndSize
{
    public string Name;
    public int Size;
}

public static NameAndSize MaxNameAndSize(IList<NameAndSize> list)
{
    if (list.Count() == 0)
    {
        return default(NameAndSize);
    }

    var maxSoFar = list[0];
    foreach (var item in list)
    {
        if (item.Size > maxSoFar.Size)
        {
            maxSoFar = item;
        }
    }
    return maxSoFar;
}
```

Doing this in LINQ seems hard to do efficiently (that is, in one pass), and has come up as a Stack Overflow question. Jon Skeet event wrote an article about it.

Again, fold to the rescue!

And here's the C# code using `Aggregate` :

```
public class NameAndSize
{
    public string Name;
    public int Size;
}

public static NameAndSize MaxNameAndSize(IList<NameAndSize> list)
{
    if (!list.Any())
    {
        return default(NameAndSize);
    }

    var initialValue = list[0];
    Func<NameAndSize, NameAndSize, NameAndSize> action =
        (maxSoFar, x) => x.Size > maxSoFar.Size ? x : maxSoFar;
    return list.Aggregate(initialValue, action);
}
```

Note that this C# version returns null for an empty list. That seems dangerous -- so what should happen instead? Throwing an exception? That doesn't seem right either.

Here's the F# code using fold:

```
type NameAndSize= {Name:string;Size:int}

let maxNameAndSize list =

    let innerMaxNameAndSize initialValue rest =
        let action maxSoFar x = if maxSoFar.Size < x.Size then x else maxSoFar
        rest |> List.fold action initialValue

    // handle empty lists
    match list with
    | [] ->
        None
    | first::rest ->
        let max = innerMaxNameAndSize first rest
        Some max
```

The F# code has two parts:

- the `innerMaxNameAndSize` function is similar to what we have seen before.
- the second bit, `match list with`, branches on whether the list is empty or not. With an empty list, it returns a `None`, and in the non-empty case, it returns a `Some`. Doing this guarantees that the caller of the function has to handle both cases.

And a test:

```
//test
let list = [
    {Name="Alice"; Size=10}
    {Name="Bob"; Size=1}
    {Name="Carol"; Size=12}
    {Name="David"; Size=5}
    ]
maxNameAndSize list
maxNameAndSize []
```

Actually, I didn't need to write this at all, because F# already has a `maxBy` function!

```
// use the built in function
list |> List.maxBy (fun item -> item.Size)
[] |> List.maxBy (fun item -> item.Size)
```

But as you can see, it doesn't handle empty lists well. Here's a version that wraps the `maxBy` safely.

```
let maxNameAndSize list =
    match list with
    | [] ->
        None
    | _ ->
        let max = list |> List.maxBy (fun item -> item.Size)
        Some max
```

Using functions as building blocks

A well-known principle of good design is to create a set of basic operations and then combine these building blocks in various ways to build up more complex behaviors. In object-oriented languages, this goal gives rise to a number of implementation approaches such as "fluent interfaces", "strategy pattern", "decorator pattern", and so on. In F#, they are all done the same way, via function composition.

Let's start with a simple example using integers. Say that we have created some basic functions to do arithmetic:

```
// building blocks
let add2 x = x + 2
let mult3 x = x * 3
let square x = x * x

// test
[1..10] |> List.map add2 |> printfn "%A"
[1..10] |> List.map mult3 |> printfn "%A"
[1..10] |> List.map square |> printfn "%A"
```

Now we want to create new functions that build on these:

```
// new composed functions
let add2ThenMult3 = add2 >> mult3
let mult3ThenSquare = mult3 >> square
```

The " `>>` " operator is the composition operator. It means: do the first function, and then do the second.

Note how concise this way of combining functions is. There are no parameters, types or other irrelevant noise.

To be sure, the examples could also have been written less concisely and more explicitly as:

```
let add2ThenMult3 x = mult3 (add2 x)
let mult3ThenSquare x = square (mult3 x)
```

But this more explicit style is also a bit more cluttered:

- In the explicit style, the x parameter and the parentheses must be added, even though they don't add to the meaning of the code.

- And in the explicit style, the functions are written back-to-front from the order they are applied. In my example of `add2ThenMult3` I want to add 2 first, and then multiply. The `add2 >> mult3` syntax makes this visually clearer than `mult3(add2 x)`.

Now let's test these compositions:

```
// test
add2ThenMult3 5
mult3ThenSquare 5
[1..10] |> List.map add2ThenMult3 |> printfn "%A"
[1..10] |> List.map mult3ThenSquare |> printfn "%A"
```

Extending existing functions

Now say that we want to decorate these existing functions with some logging behavior. We can compose these as well, to make a new function with the logging built in.

```
// helper functions;
let logMsg msg x = printf "%s%i" msg x; x     //without linefeed
let logMsgN msg x = printfn "%s%i" msg x; x   //with linefeed

// new composed function with new improved logging!
let mult3ThenSquareLogged =
   logMsg "before="
   >> mult3
   >> logMsg " after mult3="
   >> square
   >> logMsgN " result="

// test
mult3ThenSquareLogged 5
[1..10] |> List.map mult3ThenSquareLogged //apply to a whole list
```

Our new function, `mult3ThenSquareLogged`, has an ugly name, but it is easy to use and nicely hides the complexity of the functions that went into it. You can see that if you define your building block functions well, this composition of functions can be a powerful way to get new functionality.

But wait, there's more! Functions are first class entities in F#, and can be acted on by any other F# code. Here is an example of using the composition operator to collapse a list of functions into a single operation.

```
let listOfFunctions = [
   mult3;
   square;
   add2;
   logMsgN "result=";
   ]

// compose all functions in the list into a single one
let allFunctions = List.reduce (>>) listOfFunctions

//test
allFunctions 5
```

Mini languages

Domain-specific languages (DSLs) are well recognized as a technique to create more readable and concise code. The functional approach is very well suited for this.

If you need to, you can go the route of having a full "external" DSL with its own lexer, parser, and so on, and there are various toolsets for F# that make this quite straightforward.

But in many cases, it is easier to stay within the syntax of F#, and just design a set of "verbs" and "nouns" that encapsulate the behavior we want.

The ability to create new types concisely and then match against them makes it very easy to set up fluent interfaces quickly. For example, here is a little function that calculates dates using a simple vocabulary. Note that two new enum-style types are defined just for this one function.

```
// set up the vocabulary
type DateScale = Hour | Hours | Day | Days | Week | Weeks
type DateDirection = Ago | Hence

// define a function that matches on the vocabulary
let getDate interval scale direction =
    let absHours = match scale with
                   | Hour | Hours -> 1 * interval
                   | Day | Days -> 24 * interval
                   | Week | Weeks -> 24 * 7 * interval
    let signedHours = match direction with
                      | Ago -> -1 * absHours
                      | Hence ->  absHours
    System.DateTime.Now.AddHours(float signedHours)

// test some examples
let example1 = getDate 5 Days Ago
let example2 = getDate 1 Hour Hence

// the C# equivalent would probably be more like this:
// getDate().Interval(5).Days().Ago()
// getDate().Interval(1).Hour().Hence()
```

The example above only has one "verb", using lots of types for the "nouns".

The following example demonstrates how you might build the functional equivalent of a fluent interface with many "verbs".

Say that we are creating a drawing program with various shapes. Each shape has a color, size, label and action to be performed when clicked, and we want a fluent interface to configure each shape.

Here is an example of what a simple method chain for a fluent interface in C# might look like:

```
FluentShape.Default
   .SetColor("red")
   .SetLabel("box")
   .OnClick( s => Console.Write("clicked") );
```

Now the concept of "fluent interfaces" and "method chaining" is really only relevant for object-oriented design. In a functional language like F#, the nearest equivalent would be the use of the pipeline operator to chain a set of functions together.

Let's start with the underlying Shape type:

```
// create an underlying type
type FluentShape = {
    label : string;
    color : string;
    onClick : FluentShape->FluentShape // a function type
    }
```

We'll add some basic functions:

```
let defaultShape =
    {label=""; color=""; onClick=fun shape->shape}

let click shape =
    shape.onClick shape

let display shape =
    printfn "My label=%s and my color=%s" shape.label shape.color
    shape   //return same shape
```

For "method chaining" to work, every function should return an object that can be used next in the chain. So you will see that the " `display` " function returns the shape, rather than nothing.

Next we create some helper functions which we expose as the "mini-language", and will be used as building blocks by the users of the language.

```
let setLabel label shape =
   {shape with FluentShape.label = label}

let setColor color shape =
   {shape with FluentShape.color = color}

//add a click action to what is already there
let appendClickAction action shape =
   {shape with FluentShape.onClick = shape.onClick >> action}
```

Notice that `appendClickAction` takes a function as a parameter and composes it with the existing click action. As you start getting deeper into the functional approach to reuse, you start seeing many more "higher order functions" like this, that is, functions that act on other functions. Combining functions like this is one of the keys to understanding the functional way of programming.

Now as a user of this "mini-language", I can compose the base helper functions into more complex functions of my own, creating my own function library. (In C# this kind of thing might be done using extension methods.)

```
// Compose two "base" functions to make a compound function.
let setRedBox = setColor "red" >> setLabel "box"

// Create another function by composing with previous function.
// It overrides the color value but leaves the label alone.
let setBlueBox = setRedBox >> setColor "blue"

// Make a special case of appendClickAction
let changeColorOnClick color = appendClickAction (setColor color)
```

I can then combine these functions together to create objects with the desired behavior.

```
//setup some test values
let redBox = defaultShape |> setRedBox
let blueBox = defaultShape |> setBlueBox

// create a shape that changes color when clicked
redBox
    |> display
    |> changeColorOnClick "green"
    |> click
    |> display  // new version after the click

// create a shape that changes label and color when clicked
blueBox
    |> display
    |> appendClickAction (setLabel "box2" >> setColor "green")
    |> click
    |> display  // new version after the click
```

In the second case, I actually pass two functions to `appendClickAction`, but I compose them into one first. This kind of thing is trivial to do with a well structured functional library, but it is quite hard to do in C# without having lambdas within lambdas.

Here is a more complex example. We will create a function "`showRainbow`" that, for each color in the rainbow, sets the color and displays the shape.

```
let rainbow =
    ["red";"orange";"yellow";"green";"blue";"indigo";"violet"]

let showRainbow =
    let setColorAndDisplay color = setColor color >> display
    rainbow
    |> List.map setColorAndDisplay
    |> List.reduce (>>)

// test the showRainbow function
defaultShape |> showRainbow
```

Notice that the functions are getting more complex, but the amount of code is still quite small. One reason for this is that the function parameters can often be ignored when doing function composition, which reduces visual clutter. For example, the "`showRainbow`" function does take a shape as a parameter, but it is not explicitly shown! This elision of parameters is called "point-free" style and will be discussed further in the "thinking functionally" series

Pattern matching for conciseness

So far we have seen the pattern matching logic in the `match..with` expression, where it seems to be just a switch/case statement. But in fact pattern matching is much more general ? it can compare expressions in a number of ways, matching on values, conditions, and types, and then assign or extract values, all at the same time.

Pattern matching will be discussed in depth in later posts, but to start with, here is a little taster of one way that it aids conciseness. We'll look at the way pattern matching is used for binding values to expressions (the functional equivalent of assigning to variables).

In the following examples, we are binding to the internal members of tuples and lists directly:

```
//matching tuples directly
let firstPart, secondPart, _ =  (1,2,3)  // underscore means ignore

//matching lists directly
let elem1::elem2::rest = [1..10]       // ignore the warning for now

//matching lists inside a match..with
let listMatcher aList =
    match aList with
    | [] -> printfn "the list is empty"
    | [firstElement] -> printfn "the list has one element %A " firstElement
    | [first; second] -> printfn "list is %A and %A" first second
    | _ -> printfn "the list has more than two elements"

listMatcher [1;2;3;4]
listMatcher [1;2]
listMatcher [1]
listMatcher []
```

You can also bind values to the inside of complex structures such as records. In the following example, we will create an " `Address` " type, and then a " `Customer` " type which contains an address. Next, we will create a customer value, and then match various properties against it.

```
// create some types
type Address = { Street: string; City: string; }
type Customer = { ID: int; Name: string; Address: Address}

// create a customer
let customer1 = { ID = 1; Name = "Bob";
      Address = {Street="123 Main"; City="NY" } }

// extract name only
let { Name=name1 } =  customer1
printfn "The customer is called %s" name1

// extract name and id
let { ID=id2; Name=name2; } =  customer1
printfn "The customer called %s has id %i" name2 id2

// extract name and address
let { Name=name3;  Address={Street=street3}  } =  customer1
printfn "The customer is called %s and lives on %s" name3 street3
```

In the last example, note how we could reach right into the `Address` substructure and pull out the street as well as the customer name.

This ability to process a nested structure, extract only the fields you want, and assign them to values, all in a single step, is very useful. It removes quite a bit of coding drudgery, and is another factor in the conciseness of typical F# code.

Convenience

In the next set of posts, we will explore a few more features of F# that I have grouped under the theme of "convenience". These features do not necessarily result in more concise code, but they do remove much of the drudgery and boilerplate code that would be needed in C#.

- **Useful "out-of-the-box" behavior for types**. Most types that you create will immediately have some useful behavior, such as immutability and built-in equality ? functionality that has to be explicitly coded for in C#.
- **All functions are "interfaces"**, meaning that many of the roles that interfaces play in object-oriented design are implicit in the way that functions work. And similarly, many object-oriented design patterns are unnecessary or trivial within a functional paradigm.
- **Partial application**. Complicated functions with many parameters can have some of the parameters fixed or "baked in" and yet leave other parameters open.
- **Active patterns**. Active patterns are a special kind of pattern where the pattern can be matched or detected dynamically, rather than statically. They are great for simplifying frequently used parsing and grouping behaviors.

Out-of-the-box behavior for types

One nice thing about F# is that most types immediately have some useful "out-of-the-box" behavior such as immutability and built-in equality, functionality that often has to be explicitly coded for in C#.

By "most" F# types, I mean the core "structural" types such as tuples, records, unions, options, lists, etc. Classes and some other types have been added to help with .NET integration, but lose some of the power of the structural types.

This built-in functionality for these core types includes:

- Immutability
- Pretty printing when debugging
- Equality
- Comparisons

Each of these is addressed below.

F# types have built-in immutability

In C# and Java, it is has become good practice to create immutable classes whenever possible. In F#, you get this for free.

Here is an immutable type in F#:

```
type PersonalName = {FirstName:string; LastName:string}
```

And here is how the same type is typically coded in C#:

```
class ImmutablePersonalName
{
    public ImmutablePersonalName(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string LastName { get; private set; }
}
```

That's 10 lines to do the same thing as 1 line of F#.

Most F# types have built-in pretty printing

In F#, you don't have to override `ToString()` for most types -- you get pretty printing for free!

You have probably already seen this when running the earlier examples. Here is another simple example:

```
type USAddress =
   {Street:string; City:string; State:string; Zip:string}
type UKAddress =
   {Street:string; Town:string; PostCode:string}
type Address = US of USAddress | UK of UKAddress
type Person =
   {Name:string; Address:Address}

let alice = {
   Name="Alice";
   Address=US {Street="123 Main";City="LA";State="CA";Zip="91201"}}
let bob = {
   Name="Bob";
   Address=UK {Street="221b Baker St";Town="London";PostCode="NW1 6XE"}}

printfn "Alice is %A" alice
printfn "Bob is %A" bob
```

The output is:

```
Alice is {Name = "Alice";
 Address = US {Street = "123 Main";
               City = "LA";
               State = "CA";
               Zip = "91201";};}
```

Most F# types have built-in structural equality

In C#, you often have to implement the `IEquatable` interface so that you can test for equality between objects. This is needed when using objects for Dictionary keys, for example.

In F#, you get this for free with most F# types. For example, using the `PersonalName` type from above, we can compare two names straight away.

```
type PersonalName = {FirstName:string; LastName:string}
let alice1 = {FirstName="Alice"; LastName="Adams"}
let alice2 = {FirstName="Alice"; LastName="Adams"}
let bob1 = {FirstName="Bob"; LastName="Bishop"}

//test
printfn "alice1=alice2 is %A" (alice1=alice2)
printfn "alice1=bob1 is %A" (alice1=bob1)
```

Most F# types are automatically comparable

In C#, you often have to implement the `IComparable` interface so that you can sort objects.

Again, in F#, you get this for free with most F# types. For example, here is a simple definition of a deck of cards.

```
type Suit = Club | Diamond | Spade | Heart
type Rank = Two | Three | Four | Five | Six | Seven | Eight
            | Nine | Ten | Jack | Queen | King | Ace
```

We can write a function to test the comparison logic:

```
let compareCard card1 card2 =
    if card1 < card2
    then printfn "%A is greater than %A" card2 card1
    else printfn "%A is greater than %A" card1 card2
```

And let's see how it works:

```
let aceHearts = Heart, Ace
let twoHearts = Heart, Two
let aceSpades = Spade, Ace

compareCard aceHearts twoHearts
compareCard twoHearts aceSpades
```

Note that the Ace of Hearts is automatically greater than the Two of Hearts, because the "Ace" rank value comes after the "Two" rank value.

But also note that the Two of Hearts is automatically greater than the Ace of Spades, because the Suit part is compared first, and the "Heart" suit value comes after the "Spade" value.

Here's an example of a hand of cards:

```
let hand = [ Club,Ace; Heart,Three; Heart,Ace;
             Spade,Jack; Diamond,Two; Diamond,Ace ]

//instant sorting!
List.sort hand |> printfn "sorted hand is (low to high) %A"
```

And as a side benefit, you get min and max for free too!

```
List.max hand |> printfn "high card is %A"
List.min hand |> printfn "low card is %A"
```

Functions as interfaces

An important aspect of functional programming is that, in a sense, all functions are "interfaces", meaning that many of the roles that interfaces play in object-oriented design are implicit in the way that functions work.

In fact, one of the critical design maxims, "program to an interface, not an implementation", is something you get for free in F#.

To see how this works, let's compare the same design pattern in C# and F#. For example, in C# we might want to use the "decorator pattern" to enhance some core code.

Let's say that we have a calculator interface:

```
interface ICalculator
{
   int Calculate(int input);
}
```

And then a specific implementation:

```
class AddingCalculator: ICalculator
{
   public int Calculate(int input) { return input + 1; }
}
```

And then if we want to add logging, we can wrap the core calculator implementation inside a logging wrapper.

```
class LoggingCalculator: ICalculator
{
   ICalculator _innerCalculator;

   LoggingCalculator(ICalculator innerCalculator)
   {
      _innerCalculator = innerCalculator;
   }

   public int Calculate(int input)
   {
      Console.WriteLine("input is {0}", input);
      var result  = _innerCalculator.Calculate(input);
      Console.WriteLine("result is {0}", result);
      return result;
   }
}
```

So far, so straightforward. But note that, for this to work, we must have defined an interface for the classes. If there had been no `ICalculator` interface, it would be necessary to retrofit the existing code.

And here is where F# shines. In F#, you can do the same thing without having to define the interface first. Any function can be transparently swapped for any other function as long as the signatures are the same.

Here is the equivalent F# code.

```
let addingCalculator input = input + 1

let loggingCalculator innerCalculator input =
   printfn "input is %A" input
   let result = innerCalculator input
   printfn "result is %A" result
   result
```

In other words, the signature of the function *is* the interface.

Generic wrappers

Even nicer is that by default, the F# logging code can be made completely generic so that it will work for *any* function at all. Here are some examples:

```
let add1 input = input + 1
let times2 input = input * 2

let genericLogger anyFunc input =
   printfn "input is %A" input   //log the input
   let result = anyFunc input    //evaluate the function
   printfn "result is %A" result //log the result
   result                        //return the result

let add1WithLogging = genericLogger add1
let times2WithLogging = genericLogger times2
```

The new "wrapped" functions can be used anywhere the original functions could be used ? no one can tell the difference!

```
// test
add1WithLogging 3
times2WithLogging 3

[1..5] |> List.map add1WithLogging
```

Exactly the same generic wrapper approach can be used for other things. For example, here is a generic wrapper for timing a function.

```
let genericTimer anyFunc input =
   let stopwatch = System.Diagnostics.Stopwatch()
   stopwatch.Start()
   let result = anyFunc input  //evaluate the function
   printfn "elapsed ms is %A" stopwatch.ElapsedMilliseconds
   result

let add1WithTimer = genericTimer add1WithLogging

// test
add1WithTimer 3
```

The ability to do this kind of generic wrapping is one of the great conveniences of the function-oriented approach. You can take any function and create a similar function based on it. As long as the new function has exactly the same inputs and outputs as the original function, the new can be substituted for the original anywhere. Some more examples:

- It is easy to write a generic caching wrapper for a slow function, so that the value is only calculated once.
- It is also easy to write a generic "lazy" wrapper for a function, so that the inner function is only called when a result is needed

The strategy pattern

We can apply this same approach to another common design pattern, the "strategy pattern."

Let's use the familiar example of inheritance: an `Animal` superclass with `Cat` and `Dog` subclasses, each of which overrides a `MakeNoise()` method to make different noises.

In a true functional design, there are no subclasses, but instead the `Animal` class would have a `NoiseMaking` function that would be passed in with the constructor. This approach is exactly the same as the "strategy" pattern in OO design.

```
type Animal(noiseMakingStrategy) =
   member this.MakeNoise =
      noiseMakingStrategy() |> printfn "Making noise %s"

// now create a cat
let meowing() = "Meow"
let cat = Animal(meowing)
cat.MakeNoise

// .. and a dog
let woofOrBark() = if (System.DateTime.Now.Second % 2 = 0)
                   then "Woof" else "Bark"
let dog = Animal(woofOrBark)
dog.MakeNoise
dog.MakeNoise  //try again a second later
```

Note that again, we do not have to define any kind of `INoiseMakingStrategy` interface first. Any function with the right signature will work. As a consequence, in the functional model, the standard .NET "strategy" interfaces such as `IComparer`, `IFormatProvider`, and `IServiceProvider` become irrelevant.

Many other design patterns can be simplified in the same way.

Partial Application

A particularly convenient feature of F# is that complicated functions with many parameters can have some of the parameters fixed or "baked in" and yet leave other parameters open. In this post, we'll take a quick look at how this might be used in practice.

Let's start with a very simple example of how this works. We'll start with a trivial function:

```
// define a adding function
let add x y = x + y

// normal use
let z = add 1 2
```

But we can do something strange as well ? we can call the function with only one parameter!

```
let add42 = add 42
```

The result is a new function that has the "42" baked in, and now takes only one parameter instead of two! This technique is called "partial application", and it means that, for any function, you can "fix" some of the parameters and leave other ones open to be filled in later.

```
// use the new function
add42 2
add42 3
```

With that under our belt, let's revisit the generic logger that we saw earlier:

```
let genericLogger anyFunc input =
   printfn "input is %A" input   //log the input
   let result = anyFunc input    //evaluate the function
   printfn "result is %A" result //log the result
   result                        //return the result
```

Unfortunately, I have hard-coded the logging operations. Ideally, I'd like to make this more generic so that I can choose how logging is done.

Of course, F# being a functional programming language, we will do this by passing functions around.

In this case we would pass "before" and "after" callback functions to the library function, like this:

```
let genericLogger before after anyFunc input =
   before input               //callback for custom behavior
   let result = anyFunc input //evaluate the function
   after result               //callback for custom behavior
   result                     //return the result
```

You can see that the logging function now has four parameters. The "before" and "after" actions are passed in as explicit parameters as well as the function and its input. To use this in practice, we just define the functions and pass them in to the library function along with the final int parameter:

```
let add1 input = input + 1

// reuse case 1
genericLogger
    (fun x -> printf "before=%i. " x) // function to call before
    (fun x -> printfn " after=%i." x) // function to call after
    add1                              // main function
    2                                 // parameter

// reuse case 2
genericLogger
    (fun x -> printf "started with=%i " x) // different callback
    (fun x -> printfn " ended with=%i" x)
    add1                              // main function
    2                                 // parameter
```

This is a lot more flexible. I don't have to create a new function every time I want to change the behavior -- I can define the behavior on the fly.

But you might be thinking that this is a bit ugly. A library function might expose a number of callback functions and it would be inconvenient to have to pass the same functions in over and over.

Luckily, we know the solution for this. We can use partial application to fix some of the parameters. So in this case, let's define a new function which fixes the `before` and `after` functions, as well as the `add1` function, but leaves the final parameter open.

```
// define a reusable function with the "callback" functions fixed
let add1WithConsoleLogging =
    genericLogger
        (fun x -> printf "input=%i. " x)
        (fun x -> printfn " result=%i" x)
        add1
        // last parameter NOT defined here yet!
```

The new "wrapper" function is called with just an int now, so the code is much cleaner. As in the earlier example, it can be used anywhere the original `add1` function could be used without any changes.

```
add1WithConsoleLogging 2
add1WithConsoleLogging 3
add1WithConsoleLogging 4
[1..5] |> List.map add1WithConsoleLogging
```

The functional approach in C#

In a classical object-oriented approach, we would probably have used inheritance to do this kind of thing. For instance, we might have had an abstract `LoggerBase` class, with virtual methods for " `before` " and " `after` " and the function to execute. And then to implement a particular kind of behavior, we would have created a new subclass and overridden the virtual methods as needed.

But classical style inheritance is now becoming frowned upon in object-oriented design, and composition of objects is much preferred. And indeed, in "modern" C#, we would probably write the code in the same way as F#, either by using events or by passing functions in.

Here's the F# code translated into C# (note that I had to specify the types for each Action)

```
public class GenericLoggerHelper<TInput, TResult>
{
    public TResult GenericLogger(
        Action<TInput> before,
        Action<TResult> after,
        Func<TInput, TResult> aFunc,
        TInput input)
    {
        before(input);             //callback for custom behavior
        var result = aFunc(input); //do the function
        after(result);             //callback for custom behavior
        return result;
    }
}
```

And here it is in use:

```
[NUnit.Framework.Test]
public void TestGenericLogger()
{
    var sut = new GenericLoggerHelper<int, int>();
    sut.GenericLogger(
        x => Console.Write("input={0}. ", x),
        x => Console.WriteLine(" result={0}", x),
        x => x + 1,
        3);
}
```

In C#, this style of programming is required when using the LINQ libraries, but many developers have not embraced it fully to make their own code more generic and adaptable. And it's not helped by the ugly `Action<>` and `Func<>` type declarations that are required. But it can certainly make the code much more reusable.

Active patterns

F# has a special type of pattern matching called "active patterns" where the pattern can be parsed or detected dynamically. As with normal patterns, the matching and output are combined into a single step from the caller's point of view.

Here is an example of using active patterns to parse a string into an int or bool.

```
// create an active pattern
let (|Int|_|) str =
   match System.Int32.TryParse(str) with
   | (true,int) -> Some(int)
   | _ -> None

// create an active pattern
let (|Bool|_|) str =
   match System.Boolean.TryParse(str) with
   | (true,bool) -> Some(bool)
   | _ -> None
```

> You don't need to worry about the complex syntax used to define the active pattern right now ? this is just an example so that you can see how they are used.

Once these patterns have been set up, they can be used as part of a normal " `match..with` " expression.

```
// create a function to call the patterns
let testParse str =
    match str with
    | Int i -> printfn "The value is an int '%i'" i
    | Bool b -> printfn "The value is a bool '%b'" b
    | _ -> printfn "The value '%s' is something else" str

// test
testParse "12"
testParse "true"
testParse "abc"
```

You can see that from the caller's point of view, the matching with an `Int` or `Bool` is transparent, even though there is parsing going on behind the scenes.

A similar example is to use active patterns with regular expressions in order to both match on a regex pattern and return the matched value in a single step.

```
// create an active pattern
open System.Text.RegularExpressions
let (|FirstRegexGroup|_|) pattern input =
   let m = Regex.Match(input,pattern)
   if (m.Success) then Some m.Groups.[1].Value else None
```

Again, once this pattern has been set up, it can be used transparently as part of a normal match expression.

```
// create a function to call the pattern
let testRegex str =
    match str with
    | FirstRegexGroup "http://(.*?)/(.*)" host ->
           printfn "The value is a url and the host is %s" host
    | FirstRegexGroup ".*?@(.*)" host ->
           printfn "The value is an email and the host is %s" host
    | _ -> printfn "The value '%s' is something else" str

// test
testRegex "http://google.com/test"
testRegex "alice@hotmail.com"
```

And for fun, here's one more: the well-known FizzBuzz challenge written using active patterns.

```
// setup the active patterns
let (|MultOf3|_|) i = if i % 3 = 0 then Some MultOf3 else None
let (|MultOf5|_|) i = if i % 5 = 0 then Some MultOf5 else None

// the main function
let fizzBuzz i =
  match i with
  | MultOf3 & MultOf5 -> printf "FizzBuzz, "
  | MultOf3 -> printf "Fizz, "
  | MultOf5 -> printf "Buzz, "
  | _ -> printf "%i, " i

// test
[1..20] |> List.iter fizzBuzz
```

Correctness

As a programmer, you are constantly judging the code that you and others write. In an ideal world, you should be able to look at a piece of code and easily understand exactly what it does; and of course, being concise, clear and readable is a major factor in this.

But more importantly, you have to be able to convince yourself that the code *does what it is supposed to do*. As you program, you are constantly reasoning about code correctness, and the little compiler in your brain is checking the code for errors and possible mistakes.

So how can a programming language help you with this?

A modern imperative language like C# provides many ways that you are already familiar with: type checking, scoping and naming rules, access modifiers and so on. And, in recent versions, static code analysis and code contracts.

All these techniques mean that the compiler can take on a lot of the burden of checking for correctness. If you make a mistake, the compiler will warn you.

But F# has some additional features that can have a huge impact on ensuring correctness. The next few posts will be devoted to four of them:

- **Immutability**, which enables code to behave much more predictably.
- **Exhaustive pattern matching**, which traps many common errors at compile time.
- **A strict type system**, which is your friend, not your enemy. You can use the static type checking almost as an instant "compile time unit test".
- **An expressive type system** that can help you "make illegal states unrepresentable"* . We'll see how to design a real-world example that demonstrates this.

* Thanks to Yaron Minsky at Jane Street for this phrase.

Immutability

To see why immutability is important, let's start with a small example.

Here's some simple C# code that processes a list of numbers.

```
public List<int> MakeList()
{
   return new List<int> {1,2,3,4,5,6,7,8,9,10};
}

public List<int> OddNumbers(List<int> list)
{
   // some code
}

public List<int> EvenNumbers(List<int> list)
{
   // some code
}
```

Now let me test it:

```
public void Test()
{
   var odds = OddNumbers(MakeList());
   var evens = EvenNumbers(MakeList());
   // assert odds = 1,3,5,7,9 -- OK!
   // assert evens = 2,4,6,8,10 -- OK!
}
```

Everything works great, and the test passes, but I notice that I am creating the list twice ? surely I should refactor this out? So I do the refactoring, and here's the new improved version:

```
public void RefactoredTest()
{
   var list = MakeList();
   var odds = OddNumbers(list);
   var evens = EvenNumbers(list);
   // assert odds = 1,3,5,7,9 -- OK!
   // assert evens = 2,4,6,8,10 -- FAIL!
}
```

But now the test suddenly fails! Why would a refactoring break the test? Can you tell just by looking at the code?

The answer is, of course, that the list is mutable, and it is probable that the `OddNumbers` function is making destructive changes to the list as part of its filtering logic. Of course, in order to be sure, we would have to examine the code inside the `OddNumbers` function.

In other words, when I call the `OddNumbers` function, I am unintentionally creating undesirable side effects.

Is there a way to ensure that this cannot happen? Yes -- if the functions had used `IEnumerable` instead:

```
public IEnumerable<int> MakeList() {}
public List<int> OddNumbers(IEnumerable<int> list) {}
public List<int> EvenNumbers(IEnumerable <int> list) {}
```

In this case we can be confident that calling the `OddNumbers` function could not possibly have any effect on the list, and `EvenNumbers` would work correctly. What's more, we can know this *just by looking at the signatures*, without having to examine the internals of the functions. And if you try to make one of the functions misbehave by assigning to the list then you will get an error straight away, at compile time.

So `IEnumerable` can help in this case, but what if I had used a type such as `IEnumerable<Person>` instead of `IEnumerable<int>`? Could I still be as confident that the functions wouldn't have unintentional side effects?

Reasons why immutability is important

The example above shows why immutability is helpful. In fact, this is just the tip of the iceberg. There are a number of reasons why immutability is important:

- Immutable data makes the code predictable
- Immutable data is easier to work with
- Immutable data forces you to use a "transformational" approach

First, immutability makes the code **predictable**. If data is immutable, there can be no side-effects. If there are no side-effects, it is much, much, easier to reason about the correctness of the code.

And when you have two functions that work on immutable data, you don't have to worry about which order to call them in, or whether one function will mess with the input of the other function. And you have peace of mind when passing data around (for example, you

don't have to worry about using an object as a key in a hashtable and having its hash code change).

In fact, immutability is a good idea for the same reasons that global variables are a bad idea: data should be kept as local as possible and side-effects should be avoided.

Second, immutability is **easier to work with**. If data is immutable, many common tasks become much easier. Code is easier to write and easier to maintain. Fewer unit tests are needed (you only have to check that a function works in isolation), and mocking is much easier. Concurrency is much simpler, as you don't have to worry about using locks to avoid update conflicts (because there are no updates).

Finally, using immutability by default means that you start thinking differently about programming. You tend to think about **transforming** the data rather than mutating it in place.

SQL queries and LINQ queries are good examples of this "transformational" approach. In both cases, you always transform the original data through various functions (selects, filters, sorts) rather than modifying the original data.

When a program is designed using a transformation approach, the result tends to be more elegant, more modular, and more scalable. And as it happens, the transformation approach is also a perfect fit with a function-oriented paradigm.

How F# does immutability

We saw earlier that immutable values and types are the default in F#:

```
// immutable list
let list = [1;2;3;4]

type PersonalName = {FirstName:string; LastName:string}
// immutable person
let john = {FirstName="John"; LastName="Doe"}
```

Because of this, F# has a number of tricks to make life easier and to optimize the underlying code.

First, since you can't modify a data structure, you must copy it when you want to change it. F# makes it easy to copy another data structure with only the changes you want:

```
let alice = {john with FirstName="Alice"}
```

And complex data structures are implemented as linked lists or similar, so that common parts of the structure are shared.

```
// create an immutable list
let list1 = [1;2;3;4]

// prepend to make a new list
let list2 = 0::list1

// get the last 4 of the second list
let list3 = list2.Tail

// the two lists are the identical object in memory!
System.Object.ReferenceEquals(list1,list3)
```

This technique ensures that, while you might appear to have hundreds of copies of a list in your code, they are all sharing the same memory behind the scenes.

Mutable data

F# is not dogmatic about immutability; it does support mutable data with the `mutable` keyword. But turning on mutability is an explicit decision, a deviation from the default, and it is generally only needed for special cases such as optimization, caching, etc, or when dealing with the .NET libraries.

In practice, a serious application is bound to have some mutable state if it deals with messy world of user interfaces, databases, networks and so on. But F# encourages the minimization of such mutable state. You can generally still design your core business logic to use immutable data, with all the corresponding benefits.

Exhaustive pattern matching

We briefly noted earlier that when pattern matching there is a requirement to match all possible cases. This turns out be a very powerful technique to ensure correctness.

Let's compare some C# to F# again. Here's some C# code that uses a switch statement to handle different types of state.

```
enum State { New, Draft, Published, Inactive, Discontinued }
void HandleState(State state)
{
    switch (state)
    {
        case State.Inactive: // code for Inactive
            break;
        case State.Draft: // code for Draft
            break;
        case State.New: // code for New
            break;
        case State.Discontinued: // code for Discontinued
            break;
    }
}
```

This code will compile, but there is an obvious bug! The compiler couldn't see it ? can you? If you can, and you fixed it, would it stay fixed if I added another `State` to the list?

Here's the F# equivalent:

```
type State = New | Draft | Published | Inactive | Discontinued
let handleState state =
   match state with
   | Inactive -> () // code for Inactive
   | Draft -> () // code for Draft
   | New -> () // code for New
   | Discontinued -> () // code for Discontinued
```

Now try running this code. What does the compiler tell you?

The fact that exhaustive matching is always done means that certain common errors will be detected by the compiler immediately:

- A missing case (often caused when a new choice has been added due to changed requirements or refactoring).
- An impossible case (when an existing choice has been removed).

- A redundant case that could never be reached (the case has been subsumed in a previous case -- this can sometimes be non-obvious).

Now let's look at some real examples of how exhaustive matching can help you write correct code.

Avoiding nulls with the Option type

We'll start with an extremely common scenario where the caller should always check for an invalid case, namely testing for nulls. A typical C# program is littered with code like this:

```
if (myObject != null)
{
  // do something
}
```

Unfortunately, this test is not required by the compiler. All it takes is for one piece of code to forget to do this, and the program can crash. Over the years, a huge amount of programming effort has been devoted to handling nulls ? the invention of nulls has even been called a billion dollar mistake!

In pure F#, nulls cannot exist accidentally. A string or object must always be assigned to something at creation, and is immutable thereafter.

However, there are many situations where the *design intent* is to distinguish between valid and invalid values, and you require the caller to handle both cases.

In C#, this can be managed in certain situations by using nullable value types (such as `Nullable<int>`) to make the design decision clear. When a nullable is encountered the compiler will force you to be aware of it. You can then test the validity of the value before using it. But nullables do not work for standard classes (i.e. reference types), and it is easy to accidentally bypass the tests too and just call `Value` directly.

In F# there is a similar but more powerful concept to convey the design intent: the generic wrapper type called `Option` , with two choices: `Some` or `None` . The `Some` choice wraps a valid value, and `None` represents a missing value.

Here's an example where `Some` is returned if a file exists, but a missing file returns `None` .

```
let getFileInfo filePath =
   let fi = new System.IO.FileInfo(filePath)
   if fi.Exists then Some(fi) else None

let goodFileName = "good.txt"
let badFileName = "bad.txt"

let goodFileInfo = getFileInfo goodFileName // Some(fileinfo)
let badFileInfo = getFileInfo badFileName   // None
```

If we want to do anything with these values, we must always handle both possible cases.

```
match goodFileInfo with
  | Some fileInfo ->
      printfn "the file %s exists" fileInfo.FullName
  | None ->
      printfn "the file doesn't exist"

match badFileInfo with
  | Some fileInfo ->
      printfn "the file %s exists" fileInfo.FullName
  | None ->
      printfn "the file doesn't exist"
```

We have no choice about this. Not handling a case is a compile-time error, not a run-time error. By avoiding nulls and by using `Option` types in this way, F# completely eliminates a large class of null reference exceptions.

Caveat: F# does allow you to access the value without testing, just like C#, but that is considered extremely bad practice.

Exhaustive pattern matching for edge cases

Here's some C# code that creates a list by averaging pairs of numbers from an input list:

```
public IList<float> MovingAverages(IList<int> list)
{
    var averages = new List<float>();
    for (int i = 0; i < list.Count; i++)
    {
        var avg = (list[i] + list[i+1]) / 2;
        averages.Add(avg);
    }
    return averages;
}
```

It compiles correctly, but it actually has a couple of issues. Can you find them quickly? If you're lucky, your unit tests will find them for you, assuming you have thought of all the edge cases.

Now let's try the same thing in F#:

```
let rec movingAverages list =
    match list with
    // if input is empty, return an empty list
    | [] -> []
    // otherwise process pairs of items from the input
    | x::y::rest ->
        let avg = (x+y)/2.0
        //build the result by recursing the rest of the list
        avg :: movingAverages (y::rest)
```

This code also has a bug. But unlike C#, this code will not even compile until I fix it. The compiler will tell me that I haven't handled the case when I have a single item in my list. Not only has it found a bug, it has revealed a gap in the requirements: what should happen when there is only one item?

Here's the fixed up version:

```
let rec movingAverages list =
    match list with
    // if input is empty, return an empty list
    | [] -> []
    // otherwise process pairs of items from the input
    | x::y::rest ->
        let avg = (x+y)/2.0
        //build the result by recursing the rest of the list
        avg :: movingAverages (y::rest)
    // for one item, return an empty list
    | [_] -> []

// test
movingAverages [1.0]
movingAverages [1.0; 2.0]
movingAverages [1.0; 2.0; 3.0]
```

As an additional benefit, the F# code is also much more self-documenting. It explicitly describes the consequences of each case. In the C# code, it is not at all obvious what happens if a list is empty or only has one item. You would have to read the code carefully to find out.

Exhaustive pattern matching as an error handling technique

The fact that all choices must be matched can also be used as a useful alternative to throwing exceptions. For example consider the following common scenario:

- There is a utility function in the lowest tier of your app that opens a file and performs an arbitrary operation on it (that you pass in as a callback function)
- The result is then passed back up through to tiers to the top level.
- A client calls the top level code, and the result is processed and any error handling done.

In a procedural or OO language, propagating and handling exceptions across layers of code is a common problem. Top level functions are not easily able to tell the difference between an exception that they should recover from (`FileNotFound` say) vs. an exception that they needn't handle (`OutOfMemory` say). In Java, there has been an attempt to do this with checked exceptions, but with mixed results.

In the functional world, a common technique is to create a new structure to hold both the good and bad possibilities, rather than throwing an exception if the file is missing.

```
// define a "union" of two different alternatives
type Result<'a, 'b> =
    | Success of 'a  // 'a means generic type. The actual type
                     // will be determined when it is used.
    | Failure of 'b  // generic failure type as well

// define all possible errors
type FileErrorReason =
    | FileNotFound of string
    | UnauthorizedAccess of string * System.Exception

// define a low level function in the bottom layer
let performActionOnFile action filePath =
   try
      //open file, do the action and return the result
      use sr = new System.IO.StreamReader(filePath:string)
      let result = action sr  //do the action to the reader
      sr.Close()
      Success (result)        // return a Success
   with      // catch some exceptions and convert them to errors
      | :? System.IO.FileNotFoundException as ex
          -> Failure (FileNotFound filePath)
      | :? System.Security.SecurityException as ex
          -> Failure (UnauthorizedAccess (filePath,ex))
      // other exceptions are unhandled
```

The code demonstrates how `performActionOnFile` returns a `Result` object which has two alternatives: `Success` and `Failure`. The `Failure` alternative in turn has two alternatives as well: `FileNotFound` and `UnauthorizedAccess`.

Now the intermediate layers can call each other, passing around the result type without worrying what its structure is, as long as they don't access it:

```
// a function in the middle layer
let middleLayerDo action filePath =
    let fileResult = performActionOnFile action filePath
    // do some stuff
    fileResult //return

// a function in the top layer
let topLayerDo action filePath =
    let fileResult = middleLayerDo action filePath
    // do some stuff
    fileResult //return
```

Because of type inference, the middle and top layers do not need to specify the exact types returned. If the lower layer changes the type definition at all, the intermediate layers will not be affected.

Obviously at some point, a client of the top layer does want to access the result. And here is where the requirement to match all patterns is enforced. The client must handle the case with a `Failure` or else the compiler will complain. And furthermore, when handling the `Failure` branch, it must handle the possible reasons as well. In other words, special case handling of this sort can be enforced at compile time, not at runtime! And in addition the possible reasons are explicitly documented by examining the reason type.

Here is an example of a client function that accesses the top layer:

```
/// get the first line of the file
let printFirstLineOfFile filePath =
    let fileResult = topLayerDo (fun fs->fs.ReadLine()) filePath

    match fileResult with
    | Success result ->
        // note type-safe string printing with %s
        printfn "first line is: '%s'" result
    | Failure reason ->
       match reason with  // must match EVERY reason
       | FileNotFound file ->
           printfn "File not found: %s" file
       | UnauthorizedAccess (file,_) ->
           printfn "You do not have access to the file: %s" file
```

You can see that this code must explicitly handle the `Success` and `Failure` cases, and then for the failure case, it explicitly handles the different reasons. If you want to see what happens if it does not handle one of the cases, try commenting out the line that handles `UnauthorizedAccess` and see what the compiler says.

Now it is not required that you always handle all possible cases explicitly. In the example below, the function uses the underscore wildcard to treat all the failure reasons as one. This can considered bad practice if we want to get the benefits of the strictness, but at least it is clearly done.

```
/// get the length of the text in the file
let printLengthOfFile filePath =
   let fileResult =
     topLayerDo (fun fs->fs.ReadToEnd().Length) filePath

   match fileResult with
   | Success result ->
      // note type-safe int printing with %i
      printfn "length is: %i" result
   | Failure _ ->
      printfn "An error happened but I don't want to be specific"
```

Now let's see all this code work in practice with some interactive tests.

First set up a good file and a bad file.

```
/// write some text to a file
let writeSomeText filePath someText =
    use writer = new System.IO.StreamWriter(filePath:string)
    writer.WriteLine(someText:string)
    writer.Close()

let goodFileName = "good.txt"
let badFileName = "bad.txt"

writeSomeText goodFileName "hello"
```

And now test interactively:

```
printFirstLineOfFile goodFileName
printLengthOfFile goodFileName

printFirstLineOfFile badFileName
printLengthOfFile badFileName
```

I think you can see that this approach is very attractive:

- Functions return error types for each expected case (such as `FileNotFound`), but the handling of these types does not need to make the calling code ugly.
- Functions continue to throw exceptions for unexpected cases (such as `OutOfMemory`), which will generally be caught and logged at the top level of the program.

This technique is simple and convenient. Similar (and more generic) approaches are standard in functional programming.

It is feasible to use this approach in C# too, but it is normally impractical, due to the lack of union types and the lack of type inference (we would have to specify generic types everywhere).

Exhaustive pattern matching as a change management tool

Finally, exhaustive pattern matching is a valuable tool for ensuring that code stays correct as requirements change, or during refactoring.

Let's say that the requirements change and we need to handle a third type of error: "Indeterminate". To implement this new requirement, change the first `Result` type as follows, and re-evaluate all the code. What happens?

```
type Result<'a, 'b> =
    | Success of 'a
    | Failure of 'b
    | Indeterminate
```

Or sometimes a requirements change will remove a possible choice. To emulate this, change the first `Result` type to eliminate all but one of the choices.

```
type Result<'a> =
    | Success of 'a
```

Now re-evaluate the rest of the code. What happens now?

This is very powerful! When we adjust the choices, we immediately know all the places which need to be fixed to handle the change. This is another example of the power of statically checked type errors. It is often said about functional languages like F# that "if it compiles, it must be correct".

Using the type system to ensure correct code

You are familiar with static type checking through languages such as C# and Java. In these languages, the type checking is straightforward but rather crude, and can be seen as an annoyance compared with the freedom of dynamic languages such as Python and Ruby.

But in F# the type system is your friend, not your enemy. You can use static type checking almost as an instant unit test ? making sure that your code is correct at compile time.

In the earlier posts we have already seen some of the things that you can do with the type system in F#:

- The types and their associated functions provide an abstraction to model the problem domain. Because creating types is so easy, there is rarely an excuse to avoid designing them as needed for a given problem, and unlike C# classes it is hard to create "kitchen-sink" types that do everything.
- Well defined types aid in maintenance. Since F# uses type inference, you can normally rename or restructure types easily without using a refactoring tool. And if the type is changed in an incompatible way, this will almost certainly create compile-time errors that aid in tracking down any problems.
- Well named types provide instant documentation about their roles in the program (and this documentation can never be out of date).

In this post and the next we will focus on using the type system as an aid to writing correct code. I will demonstrate that you can create designs such that, if your code actually compiles, it will almost certainly work as designed.

Using standard type checking

In C#, you use the compile-time checks to validate your code without even thinking about it. For example, would you give up `List<string>` for a plain `List`? Or give up `Nullable<int>` and be forced to used `object` with casting? Probably not.

But what if you could have even more fine-grained types? You could have even better compile-time checks. And this is exactly what F# offers.

The F# type checker is not that much stricter than the C# type checker. But because it is so easy to create new types without clutter, you can represent the domain better, and, as a useful side-effect, avoid many common errors.

Here is a simple example:

```
//define a "safe" email address type
type EmailAddress = EmailAddress of string

//define a function that uses it
let sendEmail (EmailAddress email) =
   printfn "sent an email to %s" email

//try to send one
let aliceEmail = EmailAddress "alice@example.com"
sendEmail aliceEmail

//try to send a plain string
sendEmail "bob@example.com"   //error
```

By wrapping the email address in a special type, we ensure that normal strings cannot be used as arguments to email specific functions. (In practice, we would also hide the constructor of the `EmailAddress` type as well, to ensure that only valid values could be created in the first place.)

There is nothing here that couldn't be done in C#, but it would be quite a lot of work to create a new value type just for this one purpose, so in C#, it is easy to be lazy and just pass strings around.

Additional type safety features in F#

Before moving on to the major topic of "designing for correctness", let's see a few of the other minor, but cool, ways that F# is type-safe.

Type-safe formatting with printf

Here is a minor feature that demonstrates one of the ways that F# is more type-safe than C#, and how the F# compiler can catch errors that would only be detected at runtime in C#.

Try evaluating the following and look at the errors generated:

```
let printingExample =
   printf "an int %i" 2                        // ok
   printf "an int %i" 2.0                      // wrong type
   printf "an int %i" "hello"                  // wrong type
   printf "an int %i"                          // missing param

   printf "a string %s" "hello"                // ok
   printf "a string %s" 2                      // wrong type
   printf "a string %s"                        // missing param
   printf "a string %s" "he" "lo"              // too many params

   printf "an int %i and string %s" 2 "hello"  // ok
   printf "an int %i and string %s" "hello" 2  // wrong type
   printf "an int %i and string %s" 2          // missing param
```

Unlike C#, the compiler analyses the format string and determines what the number and types of the arguments are supposed to be.

This can be used to constrain the types of parameters without explicitly having to specify them. So for example, in the code below, the compiler can deduce the types of the arguments automatically.

```
let printAString x = printf "%s" x
let printAnInt x = printf "%i" x

// the result is:
// val printAString : string -> unit  //takes a string parameter
// val printAnInt : int -> unit       //takes an int parameter
```

Units of measure

F# has the ability to define units of measure and associate them with floats. The unit of measure is then "attached" to the float as a type and prevents mixing different types. This is another feature that can be very handy if you need it.

```
// define some measures
[<Measure>]
type cm

[<Measure>]
type inches

[<Measure>]
type feet =
   // add a conversion function
   static member toInches(feet : float<feet>) : float<inches> =
      feet * 12.0<inches/feet>

// define some values
let meter = 100.0<cm>
let yard = 3.0<feet>

//convert to different measure
let yardInInches = feet.toInches(yard)

// can't mix and match!
yard + meter

// now define some currencies
[<Measure>]
type GBP

[<Measure>]
type USD

let gbp10 = 10.0<GBP>
let usd10 = 10.0<USD>
gbp10 + gbp10             // allowed: same currency
gbp10 + usd10             // not allowed: different currency
gbp10 + 1.0               // not allowed: didn't specify a currency
gbp10 + 1.0<_>            // allowed using wildcard
```

Type-safe equality

One final example. In C# any class can be equated with any other class (using reference equality by default). In general, this is a bad idea! For example, you shouldn't really be able to compare a string with a person at all.

Here is some C# code which is perfectly valid and compiles fine:

```
using System;
var obj = new Object();
var ex = new Exception();
var b = (obj == ex);
```

If we write the identical code in F#, we get a compile-time error:

```
open System
let obj = new Object()
let ex = new Exception()
let b = (obj = ex)
```

Chances are, if you are testing equality between two different types, you are doing something wrong.

In F#, you can even stop a type being compared at all! This is not as silly as it seems. For some types, there may not be a useful default, or you may want to force equality to be based on a specific field rather than the object as whole.

Here is an example of this:

```
// deny comparison
[<NoEquality; NoComparison>]
type CustomerAccount = {CustomerAccountId: int}

let x = {CustomerAccountId = 1}

x = x       // error!
x.CustomerAccountId = x.CustomerAccountId // no error
```

Worked example: Designing for correctness

In this post, we'll see how you can design for correctness (or at least, for the requirements as you currently understand them), by which I mean that a client of a well designed model will not be able to put the system into an illegal state ? a state that doesn't meet the requirements. You literally cannot create incorrect code because the compiler will not let you.

For this to work, we do have to spend some time up front thinking about design and making an effort to encode the requirements into the types that you use. If you just use strings or lists for all your data structures, you will not get any benefit from the type checking.

We'll use a simple example. Let's say that you are designing an e-commerce site which has a shopping cart and you are given the following requirements.

- You can only pay for a cart once.
- Once a cart is paid for, you cannot change the items in it.
- Empty carts cannot be paid for.

A bad design in C#

In C#, we might think that this is simple enough and dive straight into coding. Here is a straightforward implementation in C# that seems OK at first glance.

```
public class NaiveShoppingCart<TItem>
{
   private List<TItem> items;
   private decimal paidAmount;

   public NaiveShoppingCart()
   {
      this.items = new List<TItem>();
      this.paidAmount = 0;
   }

   /// Is cart paid for?
   public bool IsPaidFor { get { return this.paidAmount > 0; } }

   /// Readonly list of items
   public IEnumerable<TItem> Items { get {return this.items; } }

   /// add item only if not paid for
   public void AddItem(TItem item)
   {
      if (!this.IsPaidFor)
      {
         this.items.Add(item);
      }
   }

   /// remove item only if not paid for
   public void RemoveItem(TItem item)
   {
      if (!this.IsPaidFor)
      {
         this.items.Remove(item);
      }
   }

   /// pay for the cart
   public void Pay(decimal amount)
   {
      if (!this.IsPaidFor)
      {
         this.paidAmount = amount;
      }
   }
}
```

Unfortunately, it's actually a pretty bad design:

- One of the requirements is not even met. Can you see which one?
- It has a major design flaw, and a number of minor ones. Can you see what they are?

So many problems in such a short piece of code!

What would happen if we had even more complicated requirements and the code was thousands of lines long? For example, the fragment that is repeated everywhere:

```
if (!this.IsPaidFor) { do something }
```

looks like it will be quite brittle if requirements change in some methods but not others.

Before you read the next section, think for a minute how you might better implement the requirements above in C#, with these additional requirements:

- If you try to do something that is not allowed in the requirements, you will get a *compile time error*, not a run time error. For example, you must create a design such that you cannot even call the `RemoveItem` method from an empty cart.
- The contents of the cart in any state should be immutable. The benefit of this is that if I am in the middle of paying for a cart, the cart contents can't change even if some other process is adding or removing items at the same time.

A correct design in F#

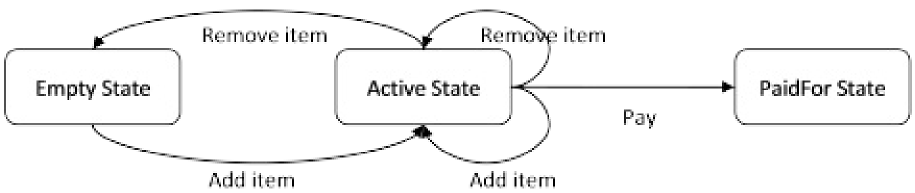
Let's step back and see if we can come up with a better design. Looking at these requirements, it's obvious that we have a simple state machine with three states and some state transitions:

- A Shopping Cart can be Empty, Active or PaidFor
- When you add an item to an Empty cart, it becomes Active
- When you remove the last item from an Active cart, it becomes Empty
- When you pay for an Active cart, it becomes PaidFor

And now we can add the business rules to this model:

- You can add an item only to carts that are Empty or Active
- You can remove an item only from carts that are Active
- You can only pay for carts that are Active

Here is the state diagram:



It's worth noting that these kinds of state-oriented models are very common in business systems. Product development, customer relationship management, order processing, and other workflows can often be modeled this way.

Now we have the design, we can reproduce it in F#:

```
type CartItem = string    // placeholder for a more complicated type

type EmptyState = NoItems // don't use empty list! We want to
                          // force clients to handle this as a
                          // separate case. E.g. "you have no
                          // items in your cart"

type ActiveState = { UnpaidItems : CartItem list; }
type PaidForState = { PaidItems : CartItem list;
                      Payment : decimal}

type Cart =
    | Empty of EmptyState
    | Active of ActiveState
    | PaidFor of PaidForState
```

We create a type for each state, and `Cart` type that is a choice of any one of the states. I have given everything a distinct name (e.g. `PaidItems` and `UnpaidItems` rather than just `Items`) because this helps the inference engine and makes the code more self documenting.

> This is a much longer example than the earlier ones! Don't worry too much about the F# syntax right now, but I hope that you can get the gist of the code, and see how it fits into the overall design.
>
> Also, do paste the snippets into a script file and evaluate them for yourself as they come up.

Next we can create the operations for each state. The main thing to note is each operation will always take one of the States as input and return a new Cart. That is, you start off with a particular known state, but you return a `Cart` which is a wrapper for a choice of three possible states.

```
// =============================
// operations on empty state
// =============================

let addToEmptyState item =
   // returns a new Active Cart
   Cart.Active {UnpaidItems=[item]}

// =============================
// operations on active state
// =============================

let addToActiveState state itemToAdd =
   let newList = itemToAdd :: state.UnpaidItems
   Cart.Active {state with UnpaidItems=newList }

let removeFromActiveState state itemToRemove =
   let newList = state.UnpaidItems
                 |> List.filter (fun i -> i<>itemToRemove)

   match newList with
   | [] -> Cart.Empty NoItems
   | _ -> Cart.Active {state with UnpaidItems=newList}

let payForActiveState state amount =
   // returns a new PaidFor Cart
   Cart.PaidFor {PaidItems=state.UnpaidItems; Payment=amount}
```

Next, we attach the operations to the states as methods

```
type EmptyState with
   member this.Add = addToEmptyState

type ActiveState with
   member this.Add = addToActiveState this
   member this.Remove = removeFromActiveState this
   member this.Pay = payForActiveState this
```

And we can create some cart level helper methods as well. At the cart level, we have to explicitly handle each possibility for the internal state with a `match..with` expression.

```
let addItemToCart cart item =
   match cart with
   | Empty state -> state.Add item
   | Active state -> state.Add item
   | PaidFor state ->
       printfn "ERROR: The cart is paid for"
       cart

let removeItemFromCart cart item =
   match cart with
   | Empty state ->
      printfn "ERROR: The cart is empty"
      cart   // return the cart
   | Active state ->
      state.Remove item
   | PaidFor state ->
      printfn "ERROR: The cart is paid for"
      cart   // return the cart

let displayCart cart  =
   match cart with
   | Empty state ->
      printfn "The cart is empty"   // can't do state.Items
   | Active state ->
      printfn "The cart contains %A unpaid items"
                                          state.UnpaidItems
   | PaidFor state ->
      printfn "The cart contains %A paid items. Amount paid: %f"
                                    state.PaidItems state.Payment

type Cart with
   static member NewCart = Cart.Empty NoItems
   member this.Add = addItemToCart this
   member this.Remove = removeItemFromCart this
   member this.Display = displayCart this
```

Testing the design

Let's exercise this code now:

```
let emptyCart = Cart.NewCart
printf "emptyCart="; emptyCart.Display

let cartA = emptyCart.Add "A"
printf "cartA="; cartA.Display
```

We now have an active cart with one item in it. Note that " `cartA` " is a completely different object from " `emptyCart` " and is in a different state.

Let's keep going:

```
let cartAB = cartA.Add "B"
printf "cartAB="; cartAB.Display

let cartB = cartAB.Remove "A"
printf "cartB="; cartB.Display

let emptyCart2 = cartB.Remove "B"
printf "emptyCart2="; emptyCart2.Display
```

So far, so good. Again, all these are distinct objects in different states,

Let's test the requirement that you cannot remove items from an empty cart:

```
let emptyCart3 = emptyCart2.Remove "B"    //error
printf "emptyCart3="; emptyCart3.Display
```

An error ? just what we want!

Now say that we want to pay for a cart. We didn't create this method at the Cart level, because we didn't want to tell the client how to handle all the cases. This method only exists for the Active state, so the client will have to explicitly handle each case and only call the `Pay` method when an Active state is matched.

First we'll try to pay for cartA.

```
//  try to pay for cartA
let cartAPaid =
    match cartA with
    | Empty _ | PaidFor _ -> cartA
    | Active state -> state.Pay 100m
printf "cartAPaid="; cartAPaid.Display
```

The result was a paid cart.

Now we'll try to pay for the emptyCart.

```
//  try to pay for emptyCart
let emptyCartPaid =
    match emptyCart with
    | Empty _ | PaidFor _ -> emptyCart
    | Active state -> state.Pay 100m
printf "emptyCartPaid="; emptyCartPaid.Display
```

Nothing happens. The cart is empty, so the Active branch is not called. We might want to raise an error or log a message in the other branches, but no matter what we do we cannot accidentally call the `Pay` method on an empty cart, because that state does not have a method to call!

The same thing happens if we accidentally try to pay for a cart that is already paid.

```
//  try to pay for cartAB
let cartABPaid =
    match cartAB with
    | Empty _ | PaidFor _ -> cartAB // return the same cart
    | Active state -> state.Pay 100m

//  try to pay for cartAB again
let cartABPaidAgain =
    match cartABPaid with
    | Empty _ | PaidFor _ -> cartABPaid  // return the same cart
    | Active state -> state.Pay 100m
```

You might argue that the client code above might not be representative of code in the real world ? it is well-behaved and already dealing with the requirements.

So what happens if we have badly written or malicious client code that tries to force payment:

```
match cartABPaid with
| Empty state -> state.Pay 100m
| PaidFor state -> state.Pay 100m
| Active state -> state.Pay 100m
```

If we try to force it like this, we will get compile errors. There is no way the client can create code that does not meet the requirements.

Summary

We have designed a simple shopping cart model which has many benefits over the C# design.

- It maps to the requirements quite clearly. It is impossible for a client of this API to call code that doesn't meet the requirements.
- Using states means that the number of possible code paths is much smaller than the C# version, so there will be many fewer unit tests to write.
- Each function is simple enough to probably work the first time, as, unlike the C# version, there are no conditionals anywhere.

Analysis of the original C# code

Now that you have seen the F# code, we can revisit the original C# code with fresh eyes. In case you were wondering, here are my thoughts as to what is wrong with the C# shopping cart example as designed.

Requirement not met: An empty cart can still be paid for.

Major design flaw: Overloading the payment amount to be a signal for IsPaidFor means that a zero paid amount can never lock down the cart. Are you sure it would never be possible to have a cart which is paid for but free of charge? The requirements are not clear, but what if this did become a requirement later? How much code would have to be changed?

Minor design flaws: What should happen when trying to remove an item from an empty cart? And what should happen when attempting to pay for a cart that is already paid for? Should we throw exceptions in these cases, or just silently ignore them? And does it make sense that a client should be able to enumerate the items in an empty cart? And this is not thread safe as designed; so what happens if a secondary thread adds an item to the cart while a payment is being made on the main thread?

That's quite a lot of things to worry about.

The nice thing about the F# design is none of these problems can even exist. So designing this way not only ensures correct code, but it also really reduces the cognitive effort to ensure that the design is bullet proof in the first place.

Compile time checking: The original C# design mixes up all the states and transitions in a single class, which makes it very error prone. A better approach would be to create separate state classes (with a common base class say) which reduces complexity, but still, the lack of a built in "union" type means that you cannot statically verify that the code is correct. There are ways of doing "union" types in C#, but it is not idiomatic at all, while in F# it is commonplace.

Appendix: C# code for a correct solution

When faced with these requirements in C#, you might immediately think -- just create an interface!

But it is not as easy as you might think. I have written a follow up post on this to explain why: The shopping cart example in C#.

If you are interested to see what the C# code for a solution looks like, here it is below. This code meets the requirements above and guarantees correctness at *compile time*, as desired.

The key thing to note is that, because C# doesn't have union types, the implementation uses a "fold" function, a function that has three function parameters, one for each state. To use the cart, the caller passes a set of three lambdas in, and the (hidden) state determines what happens.

```
var paidCart = cartA.Do(
    // lambda for Empty state
    state => cartA,
    // lambda for Active state
    state => state.Pay(100),
    // lambda for Paid state
    state => cartA);
```

This approach means that the caller can never call the "wrong" function, such as "Pay" for the Empty state, because the parameter to the lambda will not support it. Try it and see!

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace WhyUseFsharp
{

    public class ShoppingCart<TItem>
    {

        #region ShoppingCart State classes

        /// <summary>
        /// Represents the Empty state
        /// </summary>
        public class EmptyState
        {
            public ShoppingCart<TItem> Add(TItem item)
            {
                var newItems = new[] { item };
                var newState = new ActiveState(newItems);
                return FromState(newState);
            }
        }

        /// <summary>
        /// Represents the Active state
        /// </summary>
        public class ActiveState
```

```
        {
            public ActiveState(IEnumerable<TItem> items)
            {
                Items = items;
            }

            public IEnumerable<TItem> Items { get; private set; }

            public ShoppingCart<TItem> Add(TItem item)
            {
                var newItems = new List<TItem>(Items) {item};
                var newState = new ActiveState(newItems);
                return FromState(newState);
            }

            public ShoppingCart<TItem> Remove(TItem item)
            {
                var newItems = new List<TItem>(Items);
                newItems.Remove(item);
                if (newItems.Count > 0)
                {
                    var newState = new ActiveState(newItems);
                    return FromState(newState);
                }
                else
                {
                    var newState = new EmptyState();
                    return FromState(newState);
                }
            }

            public ShoppingCart<TItem> Pay(decimal amount)
            {
                var newState = new PaidForState(Items, amount);
                return FromState(newState);
            }


        }

        /// <summary>
        /// Represents the Paid state
        /// </summary>
        public class PaidForState
        {
            public PaidForState(IEnumerable<TItem> items, decimal amount)
            {
                Items = items.ToList();
                Amount = amount;
            }

            public IEnumerable<TItem> Items { get; private set; }
            public decimal Amount { get; private set; }
```

```
        }

        #endregion ShoppingCart State classes

        //====================================
        // Execute of shopping cart proper
        //====================================

        private enum Tag { Empty, Active, PaidFor }
        private readonly Tag _tag = Tag.Empty;
        private readonly object _state;       //has to be a generic object

        /// <summary>
        /// Private ctor. Use FromState instead
        /// </summary>
        private ShoppingCart(Tag tagValue, object state)
        {
            _state = state;
            _tag = tagValue;
        }

        public static ShoppingCart<TItem> FromState(EmptyState state)
        {
            return new ShoppingCart<TItem>(Tag.Empty, state);
        }

        public static ShoppingCart<TItem> FromState(ActiveState state)
        {
            return new ShoppingCart<TItem>(Tag.Active, state);
        }

        public static ShoppingCart<TItem> FromState(PaidForState state)
        {
            return new ShoppingCart<TItem>(Tag.PaidFor, state);
        }

        /// <summary>
        /// Create a new empty cart
        /// </summary>
        public static ShoppingCart<TItem> NewCart()
        {
            var newState = new EmptyState();
            return FromState(newState);
        }

        /// <summary>
        /// Call a function for each case of the state
        /// </summary>
        /// <remarks>
        /// Forcing the caller to pass a function for each possible case means that al
l cases are handled at all times.
        /// </remarks>
        public TResult Do<TResult>(
```

```
            Func<EmptyState, TResult> emptyFn,
            Func<ActiveState, TResult> activeFn,
            Func<PaidForState, TResult> paidForyFn
            )
        {
            switch (_tag)
            {
                case Tag.Empty:
                    return emptyFn(_state as EmptyState);
                case Tag.Active:
                    return activeFn(_state as ActiveState);
                case Tag.PaidFor:
                    return paidForyFn(_state as PaidForState);
                default:
                    throw new InvalidOperationException(string.Format("Tag {0} not rec
ognized", _tag));
            }
        }

        /// <summary>
        /// Do an action without a return value
        /// </summary>
        public void Do(
            Action<EmptyState> emptyFn,
            Action<ActiveState> activeFn,
            Action<PaidForState> paidForyFn
            )
        {
            //convert the Actions into Funcs by returning a dummy value
            Do(
                state => { emptyFn(state); return 0; },
                state => { activeFn(state); return 0; },
                state => { paidForyFn(state); return 0; }
                );
        }



    }

    /// <summary>
    /// Extension methods for my own personal library
    /// </summary>
    public static class ShoppingCartExtension
    {
        /// <summary>
        /// Helper method to Add
        /// </summary>
        public static ShoppingCart<TItem> Add<TItem>(this ShoppingCart<TItem> cart, TI
tem item)
        {
            return cart.Do(
                state => state.Add(item), //empty case
```

```
                state => state.Add(item), //active case
                state => { Console.WriteLine("ERROR: The cart is paid for and items ca
nnot be added"); return cart; } //paid for case
            );
        }

        /// <summary>
        /// Helper method to Remove
        /// </summary>
        public static ShoppingCart<TItem> Remove<TItem>(this ShoppingCart<TItem> cart,
 TItem item)
        {
            return cart.Do(
                state => { Console.WriteLine("ERROR: The cart is empty and items canno
t be removed"); return cart; }, //empty case
                state => state.Remove(item), //active case
                state => { Console.WriteLine("ERROR: The cart is paid for and items ca
nnot be removed"); return cart; } //paid for case
            );
        }

        /// <summary>
        /// Helper method to Display
        /// </summary>
        public static void Display<TItem>(this ShoppingCart<TItem> cart)
        {
            cart.Do(
                state => Console.WriteLine("The cart is empty"),
                state => Console.WriteLine("The active cart contains {0} items", state
.Items.Count()),
                state => Console.WriteLine("The paid cart contains {0} items. Amount p
aid {1}", state.Items.Count(), state.Amount)
            );
        }
    }

    [NUnit.Framework.TestFixture]
    public class CorrectShoppingCartTest
    {
        [NUnit.Framework.Test]
        public void TestCart()
        {
            var emptyCart = ShoppingCart<string>.NewCart();
            emptyCart.Display();

            var cartA = emptyCart.Add("A");  //one item
            cartA.Display();

            var cartAb = cartA.Add("B");  //two items
            cartAb.Display();

            var cartB = cartAb.Remove("A"); //one item
            cartB.Display();
```

```
            var emptyCart2 = cartB.Remove("B"); //empty
            emptyCart2.Display();

            Console.WriteLine("Removing from emptyCart");
            emptyCart.Remove("B"); //error


            //  try to pay for cartA
            Console.WriteLine("paying for cartA");
            var paidCart = cartA.Do(
                state => cartA,
                state => state.Pay(100),
                state => cartA);
            paidCart.Display();

            Console.WriteLine("Adding to paidCart");
            paidCart.Add("C");

            //  try to pay for emptyCart
            Console.WriteLine("paying for emptyCart");
            var emptyCartPaid = emptyCart.Do(
                state => emptyCart,
                state => state.Pay(100),
                state => emptyCart);
            emptyCartPaid.Display();
        }
    }
}
```

Concurrency

We hear a lot about concurrency nowadays, how important it is, and how it is "the next major revolution in how we write software".

So what do we actually mean by "concurrency" and how can F# help?

The simplest definition of concurrency is just "several things happening at once, and maybe interacting with each other". It seems a trivial definition, but the key point is that most computer programs (and languages) are designed to work serially, on one thing at a time, and are not well-equipped to handle concurrency.

And even if computer programs are written to handle concurrency, there is an even more serious problem: our brains do not do well when thinking about concurrency. It is commonly acknowledged that writing code that handles concurrency is extremely hard. Or I should say, writing concurrent code that is *correct* is extremely hard! It's very easy to write concurrent code that is buggy; there might be race conditions, or operations might not be atomic, or tasks might be starved or blocked unnecessarily, and these issues are hard to find by looking at the code or using a debugger.

Before talking about the specifics of F#, let's try to classify some of the common types of concurrency scenarios that we have to deal with as developers:

- **"Concurrent Multitasking"**. This is when we have a number of concurrent tasks (e.g. processes or threads) within our direct control, and we want them to communicate with each other and share data safely.
- **"Asynchronous" programming**. This is when we initiate a conversation with a separate system outside our direct control, and then wait for it to get back to us. Common cases of this are when talking to the filesystem, a database, or the network. These situations are typically I/O bound, so you want to do something else useful while you are waiting. These types of tasks are often *non-deterministic* as well, meaning that running the same program twice might give a different result.
- **"Parallel" programming**. This is when we have a single task that we want to split into independant subtasks, and then run the subtasks in parallel, ideally using all the cores or CPUs that are available. These situations are typically CPU bound. Unlike the async tasks, parallelism is typically *deterministic*, so running the same program twice will give the same result.
- **"Reactive" programming**. This is when we do not initiate tasks ourselves, but are focused on listening for events which we then process as fast as possible. This situation occurs when designing servers, and when working with a user interface.

Of course, these are vague definitions and overlap in practice. In general, though, for all these cases, the actual implementations that address these scenarios tend to use two distinct approaches:

- If there are lots of different tasks that need to share state or resources without waiting, then use a "buffered asynchronous" design.
- If there are lots of identical tasks that do not need to share state, then use parallel tasks using "fork/join" or "divide and conquer" approaches.

F# tools for concurrent programming

F# offers a number of different approaches to writing concurrent code:

- For multitasking and asynchronous problems, F# can directly use all the usual .NET suspects, such as `Thread` `AutoResetEvent` , `BackgroundWorker` and `IAsyncResult` . But it also offers a much simpler model for all types of async IO and background task management, called "asynchronous workflows". We will look at these in the next post.

- An alternative approach for asynchronous problems is to use message queues and the "actor model" (this is the "buffered asynchronous" design mentioned above). F# has a built in implementation of the actor model called `MailboxProcessor` . I am a big proponent of designing with actors and message queues, as it decouples the various components and allows you to think serially about each one.

- For true CPU parallelism, F# has convenient library code that builds on the asynchronous workflows mentioned above, and it can also use the .NET Task Parallel Library.

- Finally, the functional approach to event handling and reactive programming is quite different from the traditional approach. The functional approach treats events as "streams" which can be filtered, split, and combined in much the the same way that LINQ handles collections. F# has built in support for this model, as well as for the standard event-driven model.

Asynchronous programming

In this post we'll have a look at a few ways to write asynchronous code in F#, and a very brief example of parallelism as well.

Traditional asynchronous programming

As noted in the previous post, F# can directly use all the usual .NET suspects, such as `Thread` `AutoResetEvent` , `BackgroundWorker` and `IAsyncResult` .

Let's see a simple example where we wait for a timer event to go off:

```
open System

let userTimerWithCallback =
    // create an event to wait on
    let event = new System.Threading.AutoResetEvent(false)

    // create a timer and add an event handler that will signal the event
    let timer = new System.Timers.Timer(2000.0)
    timer.Elapsed.Add (fun _ -> event.Set() |> ignore )

    //start
    printfn "Waiting for timer at %O" DateTime.Now.TimeOfDay
    timer.Start()

    // keep working
    printfn "Doing something useful while waiting for event"

    // block on the timer via the AutoResetEvent
    event.WaitOne() |> ignore

    //done
    printfn "Timer ticked at %O" DateTime.Now.TimeOfDay
```

This shows the use of `AutoResetEvent` as a synchronization mechanism.

- A lambda is registered with the `Timer.Elapsed` event, and when the event is triggered, the AutoResetEvent is signalled.
- The main thread starts the timer, does something else while waiting, and then blocks until the event is triggered.
- Finally, the main thread continues, about 2 seconds later.

The code above is reasonably straightforward, but does require you to instantiate an AutoResetEvent, and could be buggy if the lambda is defined incorrectly.

Introducing asynchronous workflows

F# has a built-in construct called "asynchronous workflows" which makes async code much easier to write. These workflows are objects that encapsulate a background task, and provide a number of useful operations to manage them.

Here's the previous example rewritten to use one:

```
open System
//open Microsoft.FSharp.Control  // Async.* is in this module.

let userTimerWithAsync =

    // create a timer and associated async event
    let timer = new System.Timers.Timer(2000.0)
    let timerEvent = Async.AwaitEvent (timer.Elapsed) |> Async.Ignore

    // start
    printfn "Waiting for timer at %O" DateTime.Now.TimeOfDay
    timer.Start()

    // keep working
    printfn "Doing something useful while waiting for event"

    // block on the timer event now by waiting for the async to complete
    Async.RunSynchronously timerEvent

    // done
    printfn "Timer ticked at %O" DateTime.Now.TimeOfDay
```

Here are the changes:

- the `AutoResetEvent` and lambda have disappeared, and are replaced by `let timerEvent = Control.Async.AwaitEvent (timer.Elapsed)`, which creates an `async` object directly from the event, without needing a lambda. The `ignore` is added to ignore the result.
- the `event.WaitOne()` has been replaced by `Async.RunSynchronously timerEvent` which blocks on the async object until it has completed.

That's it. Both simpler and easier to understand.

The async workflows can also be used with `IAsyncResult`, begin/end pairs, and other standard .NET methods.

For example, here's how you might do an async file write by wrapping the `IAsyncResult` generated from `BeginWrite`.

```
let fileWriteWithAsync =

    // create a stream to write to
    use stream = new System.IO.FileStream("test.txt",System.IO.FileMode.Create)

    // start
    printfn "Starting async write"
    let asyncResult = stream.BeginWrite(Array.empty,0,0,null,null)

    // create an async wrapper around an IAsyncResult
    let async = Async.AwaitIAsyncResult(asyncResult) |> Async.Ignore

    // keep working
    printfn "Doing something useful while waiting for write to complete"

    // block on the timer now by waiting for the async to complete
    Async.RunSynchronously async

    // done
    printfn "Async write completed"
```

Creating and nesting asynchronous workflows

Asynchronous workflows can also be created manually. A new workflow is created using the `async` keyword and curly braces. The braces contain a set of expressions to be executed in the background.

This simple workflow just sleeps for 2 seconds.

```
let sleepWorkflow  = async{
    printfn "Starting sleep workflow at %O" DateTime.Now.TimeOfDay
    do! Async.Sleep 2000
    printfn "Finished sleep workflow at %O" DateTime.Now.TimeOfDay
    }

Async.RunSynchronously sleepWorkflow
```

Note: the code `do! Async.Sleep 2000` is similar to `Thread.Sleep` but designed to work with asynchronous workflows.

Workflows can contain *other* async workflows nested inside them. Within the braces, the nested workflows can be blocked on by using the `let!` syntax.

```
let nestedWorkflow  = async{

    printfn "Starting parent"
    let! childWorkflow = Async.StartChild sleepWorkflow

    // give the child a chance and then keep working
    do! Async.Sleep 100
    printfn "Doing something useful while waiting "

    // block on the child
    let! result = childWorkflow

    // done
    printfn "Finished parent"
    }

// run the whole workflow
Async.RunSynchronously nestedWorkflow
```

Cancelling workflows

One very convenient thing about async workflows is that they support a built-in cancellation mechanism. No special code is needed.

Consider a simple task that prints numbers from 1 to 100:

```
let testLoop = async {
    for i in [1..100] do
        // do something
        printf "%i before.." i

        // sleep a bit
        do! Async.Sleep 10
        printfn "..after"
    }
```

We can test it in the usual way:

```
Async.RunSynchronously testLoop
```

Now let's say we want to cancel this task half way through. What would be the best way of doing it?

In C#, we would have to create flags to pass in and then check them frequently, but in F# this technique is built in, using the `CancellationToken` class.

Here an example of how we might cancel the task:

```
open System
open System.Threading

// create a cancellation source
let cancellationSource = new CancellationTokenSource()

// start the task, but this time pass in a cancellation token
Async.Start (testLoop,cancellationSource.Token)

// wait a bit
Thread.Sleep(200)

// cancel after 200ms
cancellationSource.Cancel()
```

In F#, any nested async call will check the cancellation token automatically!

In this case it was the line:

```
do! Async.Sleep(10)
```

As you can see from the output, this line is where the cancellation happened.

Composing workflows in series and parallel

Another useful thing about async workflows is that they can be easily combined in various ways: both in series and in parallel.

Let's again create a simple workflow that just sleeps for a given time:

```
// create a workflow to sleep for a time
let sleepWorkflowMs ms = async {
    printfn "%i ms workflow started" ms
    do! Async.Sleep ms
    printfn "%i ms workflow finished" ms
    }
```

Here's a version that combines two of these in series:

```
let workflowInSeries = async {
    let! sleep1 = sleepWorkflowMs 1000
    printfn "Finished one"
    let! sleep2 = sleepWorkflowMs 2000
    printfn "Finished two"
    }

#time
Async.RunSynchronously workflowInSeries
#time
```

And here's a version that combines two of these in parallel:

```
// Create them
let sleep1 = sleepWorkflowMs 1000
let sleep2 = sleepWorkflowMs 2000

// run them in parallel
#time
[sleep1; sleep2]
    |> Async.Parallel
    |> Async.RunSynchronously
#time
```

Note: The #time command toggles the timer on and off. It only works in the interactive window, so this example must be sent to the interactive window in order to work corrrectly.

We're using the `#time` option to show the total elapsed time, which, because they run in parallel, is 2 secs. If they ran in series instead, it would take 3 seconds.

Also you might see that the output is garbled sometimes because both tasks are writing to the console at the same time!

This last sample is a classic example of a "fork/join" approach, where a number of a child tasks are spawned and then the parent waits for them all to finish. As you can see, F# makes this very easy!

Example: an async web downloader

In this more realistic example, we'll see how easy it is to convert some existing code from a non-asynchronous style to an asynchronous style, and the corresponding performance increase that can be achieved.

So here is a simple URL downloader, very similar to the one we saw at the start of the series:

```
open System.Net
open System
open System.IO

let fetchUrl url =
    let req = WebRequest.Create(Uri(url))
    use resp = req.GetResponse()
    use stream = resp.GetResponseStream()
    use reader = new IO.StreamReader(stream)
    let html = reader.ReadToEnd()
    printfn "finished downloading %s" url
```

And here is some code to time it:

```
// a list of sites to fetch
let sites = ["http://www.bing.com";
             "http://www.google.com";
             "http://www.microsoft.com";
             "http://www.amazon.com";
             "http://www.yahoo.com"]

#time                     // turn interactive timer on
sites                     // start with the list of sites
|> List.map fetchUrl      // loop through each site and download
#time                     // turn timer off
```

Make a note of the time taken, and let's if we can improve on it!

Obviously the example above is inefficient -- only one web site at a time is visited. The program would be faster if we could visit them all at the same time.

So how would we convert this to a concurrent algorithm? The logic would be something like:

- Create a task for each web page we are downloading, and then for each task, the download logic would be something like:
    - Start downloading a page from a website. While that is going on, pause and let other tasks have a turn.
    - When the download is finished, wake up and continue on with the task
- Finally, start all the tasks up and let them go at it!

Unfortunately, this is quite hard to do in a standard C-like language. In C# for example, you have to create a callback for when an async task completes. Managing these callbacks is painful and creates a lot of extra support code that gets in the way of understanding the

logic. There are some elegant solutions to this, but in general, the signal to noise ratio for concurrent programming in C# is very high*.

* As of the time of this writing. Future versions of C# will have the `await` keyword, which is similar to what F# has now.

But as you can guess, F# makes this easy. Here is the concurrent F# version of the downloader code:

```
open Microsoft.FSharp.Control.CommonExtensions
                                        // adds AsyncGetResponse

// Fetch the contents of a web page asynchronously
let fetchUrlAsync url =
    async {
        let req = WebRequest.Create(Uri(url))
        use! resp = req.AsyncGetResponse()  // new keyword "use!"
        use stream = resp.GetResponseStream()
        use reader = new IO.StreamReader(stream)
        let html = reader.ReadToEnd()
        printfn "finished downloading %s" url
        }
```

Note that the new code looks almost exactly the same as the original. There are only a few minor changes.

- The change from " `use resp =` " to " `use! resp =` " is exactly the change that we talked about above -- while the async operation is going on, let other tasks have a turn.
- We also used the extension method `AsyncGetResponse` defined in the `CommonExtensions` namespace. This returns an async workflow that we can nest inside the main workflow.
- In addition the whole set of steps is contained in the " `async {...}` " wrapper which turns it into a block that can be run asynchronously.

And here is a timed download using the async version.

```
// a list of sites to fetch
let sites = ["http://www.bing.com";
             "http://www.google.com";
             "http://www.microsoft.com";
             "http://www.amazon.com";
             "http://www.yahoo.com"]

#time                      // turn interactive timer on
sites
|> List.map fetchUrlAsync  // make a list of async tasks
|> Async.Parallel          // set up the tasks to run in parallel
|> Async.RunSynchronously  // start them off
#time                      // turn timer off
```

The way this works is:

- `fetchUrlAsync` is applied to each site. It does not immediately start the download, but returns an async workflow for running later.
- To set up all the tasks to run at the same time we use the `Async.Parallel` function
- Finally we call `Async.RunSynchronously` to start all the tasks, and wait for them all to stop.

If you try out this code yourself, you will see that the async version is much faster than the sync version. Not bad for a few minor code changes! Most importantly, the underlying logic is still very clear and is not cluttered up with noise.

Example: a parallel computation

To finish up, let's have another quick look at a parallel computation again.

Before we start, I should warn you that the example code below is just to demonstrate the basic principles. Benchmarks from "toy" versions of parallelization like this are not meaningful, because any kind of real concurrent code has so many dependencies.

And also be aware that parallelization is rarely the best way to speed up your code. Your time is almost always better spent on improving your algorithms. I'll bet my serial version of quicksort against your parallel version of bubblesort any day! (For more details on how to improve performance, see the optimization series)

Anyway, with that caveat, let's create a little task that chews up some CPU. We'll test this serially and in parallel.

```
let childTask() =
    // chew up some CPU.
    for i in [1..1000] do
        for i in [1..1000] do
            do "Hello".Contains("H") |> ignore
            // we don't care about the answer!

// Test the child task on its own.
// Adjust the upper bounds as needed
// to make this run in about 0.2 sec
#time
childTask()
#time
```

Adjust the upper bounds of the loops as needed to make this run in about 0.2 seconds.

Now let's combine a bunch of these into a single serial task (using composition), and test it with the timer:

```
let parentTask =
    childTask
    |> List.replicate 20
    |> List.reduce (>>)

//test
#time
parentTask()
#time
```

This should take about 4 seconds.

Now in order to make the `childTask` parallelizable, we have to wrap it inside an `async` :

```
let asyncChildTask = async { return childTask() }
```

And to combine a bunch of asyncs into a single parallel task, we use `Async.Parallel` .

Let's test this and compare the timings:

```
let asyncParentTask =
    asyncChildTask
    |> List.replicate 20
    |> Async.Parallel

//test
#time
asyncParentTask
|> Async.RunSynchronously
#time
```

On a dual-core machine, the parallel version is about 50% faster. It will get faster in proportion to the number of cores or CPUs, of course, but sublinearly. Four cores will be faster than one core, but not four times faster.

On the other hand, as with the async web download example, a few minor code changes can make a big difference, while still leaving the code easy to read and understand. So in cases where parallelism will genuinely help, it is nice to know that it is easy to arrange.

Messages and Agents

In this post, we'll look at the message-based (or actor-based) approach to concurrency.

In this approach, when one task wants to communicate with another, it sends it a message, rather than contacting it directly. The messages are put on a queue, and the receiving task (known as an "actor" or "agent") pulls the messages off the queue one at a time to process them.

This message-based approach has been applied to many situations, from low-level network sockets (built on TCP/IP) to enterprise wide application integration systems (for example MSMQ or IBM WebSphere MQ).

From a software design point of view, a message-based approach has a number of benefits:

- You can manage shared data and resources without locks.
- You can easily follow the "single responsibility principle", because each agent can be designed to do only one thing.
- It encourages a "pipeline" model of programming with "producers" sending messages to decoupled "consumers", which has additional benefits:
  - The queue acts as a buffer, eliminating waiting on the client side.
  - It is straightforward to scale up one side or the other of the queue as needed in order to maximize throughput.
  - Errors can be handled gracefully, because the decoupling means that agents can be created and destroyed without affecting their clients.

From a practical developer's point of view, what I find most appealing about the message-based approach is that when writing the code for any given actor, you don't have to hurt your brain by thinking about concurrency. The message queue forces a "serialization" of operations that otherwise might occur concurrently. And this in turn makes it much easier to think about (and write code for) the logic for processing a message, because you can be sure that your code will be isolated from other events that might interrupt your flow.

With these advantages, it is not surprising that when a team inside Ericsson wanted to design a programming language for writing highly-concurrent telephony applications, they created one with a message-based approach, namely Erlang. Erlang has now become the poster child for the whole topic, and has created a lot of interest in implementing the same approach in other languages.

How F# implements a message-based approach

F# has a built-in agent class called `MailboxProcessor`. These agents are very lightweight compared with threads - you can instantiate tens of thousands of them at the same time.

These are similar to the agents in Erlang, but unlike the Erlang ones, they do *not* work across process boundaries, only in the same process. And unlike a heavyweight queueing system such as MSMQ, the messages are not persistent. If your app crashes, the messages are lost.

But these are minor issues, and can be worked around. In a future series, I will go into alternative implementations of message queues. The fundamental approach is the same in all cases.

Let's see a simple agent implementation in F#:

```
#nowarn "40"
let printerAgent = MailboxProcessor.Start(fun inbox->

    // the message processing function
    let rec messageLoop = async{

        // read a message
        let! msg = inbox.Receive()

        // process a message
        printfn "message is: %s" msg

        // loop to top
        return! messageLoop
        }

    // start the loop
    messageLoop
    )
```

The `MailboxProcessor.Start` function takes a simple function parameter. That function loops forever, reading messages from the queue (or "inbox") and processing them.

Note: I have added the #nowarn "40" pragma to avoid the warning "FS0040", which can be safely ignored in this case.

Here's the example in use:

```
// test it
printerAgent.Post "hello"
printerAgent.Post "hello again"
printerAgent.Post "hello a third time"
```

In the rest of this post we'll look at two slightly more useful examples:

- Managing shared state without locks
- Serialized and buffered access to shared IO

In both of these cases, a message based approach to concurrency is elegant, efficient, and easy to program.

Managing shared state

Let's look at the shared state problem first.

A common scenario is that you have some state that needs to be accessed and changed by multiple concurrent tasks or threads. We'll use a very simple case, and say that the requirements are:

- A shared "counter" and "sum" that can be incremented by multiple tasks concurrently.
- Changes to the counter and sum must be atomic -- we must guarantee that they will both be updated at the same time.

The locking approach to shared state

Using locks or mutexes is a common solution for these requirements, so let's write some code using a lock, and see how it performs.

First let's write a static `LockedCounter` class that protects the state with locks.

```
open System
open System.Threading
open System.Diagnostics

// a utility function
type Utility() =
    static let rand = new Random()

    static member RandomSleep() =
        let ms = rand.Next(1,10)
        Thread.Sleep ms

// an implementation of a shared counter using locks
type LockedCounter () =

    static let _lock = new Object()

    static let mutable count = 0
    static let mutable sum = 0

    static let updateState i =
        // increment the counters and...
        sum <- sum + i
        count <- count + 1
        printfn "Count is: %i. Sum is: %i" count sum

        // ...emulate a short delay
        Utility.RandomSleep()


    // public interface to hide the state
    static member Add i =
        // see how long a client has to wait
        let stopwatch = new Stopwatch()
        stopwatch.Start()

        // start lock. Same as C# lock{...}
        lock _lock (fun () ->

            // see how long the wait was
            stopwatch.Stop()
            printfn "Client waited %i" stopwatch.ElapsedMilliseconds

            // do the core logic
            updateState i
            )
        // release lock
```

Some notes on this code:

- This code is written using a very imperative approach, with mutable variables and locks

- The public `Add` method has explicit `Monitor.Enter` and `Monitor.Exit` expressions to get and release the lock. This is the same as the `lock{...}` statement in C#.
- We've also added a stopwatch to measure how long a client has to wait to get the lock.
- The core "business logic" is the `updateState` method, which not only updates the state, but adds a small random wait as well to emulate the time taken to do the processing.

Let's test it in isolation:

```
// test in isolation
LockedCounter.Add 4
LockedCounter.Add 5
```

Next, we'll create a task that will try to access the counter:

```
let makeCountingTask addFunction taskId  = async {
    let name = sprintf "Task%i" taskId
    for i in [1..3] do
        addFunction i
    }

// test in isolation
let task = makeCountingTask LockedCounter.Add 1
Async.RunSynchronously task
```

In this case, when there is no contention at all, the wait times are all 0.

But what happens when we create 10 child tasks that all try to access the counter at once:

```
let lockedExample5 =
    [1..10]
        |> List.map (fun i -> makeCountingTask LockedCounter.Add i)
        |> Async.Parallel
        |> Async.RunSynchronously
        |> ignore
```

Oh dear! Most tasks are now waiting quite a while. If two tasks want to update the state at the same time, one must wait for the other's work to complete before it can do its own work, which affects performance.

And if we add more and more tasks, the contention will increase, and the tasks will spend more and more time waiting rather than working.

The message-based approach to shared state

Let's see how a message queue might help us. Here's the message based version:

```
type MessageBasedCounter () =

    static let updateState (count,sum) msg =

        // increment the counters and...
        let newSum = sum + msg
        let newCount = count + 1
        printfn "Count is: %i. Sum is: %i" newCount newSum

        // ...emulate a short delay
        Utility.RandomSleep()

        // return the new state
        (newCount,newSum)

    // create the agent
    static let agent = MailboxProcessor.Start(fun inbox ->

        // the message processing function
        let rec messageLoop oldState = async{

            // read a message
            let! msg = inbox.Receive()

            // do the core logic
            let newState = updateState oldState msg

            // loop to top
            return! messageLoop newState
            }

        // start the loop
        messageLoop (0,0)
        )

    // public interface to hide the implementation
    static member Add i = agent.Post i
```

Some notes on this code:

- The core "business logic" is again in the `updateState` method, which has almost the same implementation as the earlier example, except the state is immutable, so that a new state is created and returned to the main loop.
- The agent reads messages (simple ints in this case) and then calls `updateState` method
- The public method `Add` posts a message to the agent, rather than calling the `updateState` method directly
- This code is written in a more functional way; there are no mutable variables and no locks anywhere. In fact, there is no code dealing with concurrency at all! The code only

has to focus on the business logic, and is consequently much easier to understand.

Let's test it in isolation:

```
// test in isolation
MessageBasedCounter.Add 4
MessageBasedCounter.Add 5
```

Next, we'll reuse a task we defined earlier, but calling `MessageBasedCounter.Add` instead:

```
let task = makeCountingTask MessageBasedCounter.Add 1
Async.RunSynchronously task
```

Finally let's create 5 child tasks that try to access the counter at once.

```
let messageExample5 =
    [1..5]
        |> List.map (fun i -> makeCountingTask MessageBasedCounter.Add i)
        |> Async.Parallel
        |> Async.RunSynchronously
        |> ignore
```

We can't measure the waiting time for the clients, because there is none!

Shared IO

A similar concurrency problem occurs when accessing a shared IO resource such as a file:

- If the IO is slow, the clients can spend a lot of time waiting, even without locks.
- If multiple threads write to the resource at the same time, you can get corrupted data.

Both problems can be solved by using asynchronous calls combined with buffering -- exactly what a message queue does.

In this next example, we'll consider the example of a logging service that many clients will write to concurrently. (In this trivial case, we'll just write directly to the Console.)

We'll first look at an implementation without concurrency control, and then at an implementation that uses message queues to serialize all requests.

IO without serialization

In order to make the corruption very obvious and repeatable, let's first create a "slow" console that writes each individual character in the log message and pauses for a millisecond between each character. During that millisecond, another thread could be writing as well, causing an undesirable interleaving of messages.

```
let slowConsoleWrite msg =
    msg |> String.iter (fun ch->
        System.Threading.Thread.Sleep(1)
        System.Console.Write ch
        )

// test in isolation
slowConsoleWrite "abc"
```

Next, we will create a simple task that loops a few times, writing its name each time to the logger:

```
let makeTask logger taskId = async {
    let name = sprintf "Task%i" taskId
    for i in [1..3] do
        let msg = sprintf "-%s:Loop%i-" name i
        logger msg
    }

// test in isolation
let task = makeTask slowConsoleWrite 1
Async.RunSynchronously task
```

Next, we write a logging class that encapsulates access to the slow console. It has no locking or serialization, and is basically not thread-safe:

```
type UnserializedLogger() =
    // interface
    member this.Log msg = slowConsoleWrite msg

// test in isolation
let unserializedLogger = UnserializedLogger()
unserializedLogger.Log "hello"
```

Now let's combine all these into a real example. We will create five child tasks and run them in parallel, all trying to write to the slow console.

```
let unserializedExample =
    let logger = new UnserializedLogger()
    [1..5]
        |> List.map (fun i -> makeTask logger.Log i)
        |> Async.Parallel
        |> Async.RunSynchronously
        |> ignore
```

Ouch! The output is very garbled!

Serialized IO with messages

So what happens when we replace `UnserializedLogger` with a `SerializedLogger` class that encapsulates a message queue.

The agent inside `SerializedLogger` simply reads a message from its input queue and writes it to the slow console. Again there is no code dealing with concurrency and no locks are used.

```
type SerializedLogger() =

    // create the mailbox processor
    let agent = MailboxProcessor.Start(fun inbox ->

        // the message processing function
        let rec messageLoop () = async{

            // read a message
            let! msg = inbox.Receive()

            // write it to the log
            slowConsoleWrite msg

            // loop to top
            return! messageLoop ()
            }

        // start the loop
        messageLoop ()
        )

    // public interface
    member this.Log msg = agent.Post msg

// test in isolation
let serializedLogger = SerializedLogger()
serializedLogger.Log "hello"
```

So now we can repeat the earlier unserialized example but using the `SerializedLogger` instead. Again, we create five child tasks and run them in parallel:

```
let serializedExample =
    let logger = new SerializedLogger()
    [1..5]
        |> List.map (fun i -> makeTask logger.Log i)
        |> Async.Parallel
        |> Async.RunSynchronously
        |> ignore
```

What a difference! This time the output is perfect.

Summary

There is much more to say about this message based approach. In a future series, I hope to go into much more detail, including discussion of topics such as:

- alternative implementations of message queues with MSMQ and TPL Dataflow.
- cancellation and out of band messages.
- error handling and retries, and handling exceptions in general.
- how to scale up and down by creating or removing child agents.
- avoiding buffer overruns and detecting starvation or inactivity.

Functional Reactive Programming

Events are everywhere. Almost every program has to handle events, whether it be button clicks in the user interface, listening to sockets in a server, or even a system shutdown notification.

And events are the basis of one of the most common OO design patterns: the "Observer" pattern.

But as we know, event handling, like concurrency in general, can be tricky to implement. Simple event logic is straightforward, but what about logic like "do something if two events happen in a row but do something different if only one event happens" or "do something if two events happen at roughly the same time". And how easy is it to combine these requirements in other, more complex ways?

Even you can successfully implement these requirements, the code tends to be spaghetti like and hard to understand, even with the best intentions.

Is there a approach that can make event handling easier?

We saw in the previous post on message queues that one of the advantages of that approach was that the requests were "serialized" making it conceptually easier to deal with.

There is a similar approach that can be used with events. The idea is to turn a series of events into an "event stream". Event streams then become quite like IEnumerables, and so the obvious next step is to treat them in much the the same way that LINQ handles collections, so that they can be filtered, mapped, split and combined.

F# has built in support for this model, as well as for the more tradition approach.

A simple event stream

Let's start with a simple example to compare the two approaches. We'll implement the classic event handler approach first.

First, we define a utility function that will:

- create a timer
- register a handler for the `Elapsed` event
- run the timer for five seconds and then stop it

Here's the code:

```
open System
open System.Threading

/// create a timer and register an event handler,
/// then run the timer for five seconds
let createTimer timerInterval eventHandler =
    // setup a timer
    let timer = new System.Timers.Timer(float timerInterval)
    timer.AutoReset <- true

    // add an event handler
    timer.Elapsed.Add eventHandler

    // return an async task
    async {
        // start timer...
        timer.Start()
        // ...run for five seconds...
        do! Async.Sleep 5000
        // ... and stop
        timer.Stop()
        }
```

Now test it interactively:

```
// create a handler. The event args are ignored
let basicHandler _ = printfn "tick %A" DateTime.Now

// register the handler
let basicTimer1 = createTimer 1000 basicHandler

// run the task now
Async.RunSynchronously basicTimer1
```

Now let's create a similar utility method to create a timer, but this time it will return an "observable" as well, which is the stream of events.

```
let createTimerAndObservable timerInterval =
    // setup a timer
    let timer = new System.Timers.Timer(float timerInterval)
    timer.AutoReset <- true

    // events are automatically IObservable
    let observable = timer.Elapsed

    // return an async task
    let task = async {
        timer.Start()
        do! Async.Sleep 5000
        timer.Stop()
        }

    // return a async task and the observable
    (task,observable)
```

And again test it interactively:

```
// create the timer and the corresponding observable
let basicTimer2 , timerEventStream = createTimerAndObservable 1000

// register that everytime something happens on the
// event stream, print the time.
timerEventStream
|> Observable.subscribe (fun _ -> printfn "tick %A" DateTime.Now)

// run the task now
Async.RunSynchronously basicTimer2
```

The difference is that instead of registering a handler directly with an event, we are "subscribing" to an event stream. Subtly different, and important.

Counting events

In this next example, we'll have a slightly more complex requirement:

```
Create a timer that ticks every 500ms.
At each tick, print the number of ticks so far and the current time.
```

To do this in a classic imperative way, we would probably create a class with a mutable counter, as below:

```
type ImperativeTimerCount() =

    let mutable count = 0

    // the event handler. The event args are ignored
    member this.handleEvent _ =
      count <- count + 1
      printfn "timer ticked with count %i" count
```

We can reuse the utility functions we created earlier to test it:

```
// create a handler class
let handler = new ImperativeTimerCount()

// register the handler method
let timerCount1 = createTimer 500 handler.handleEvent

// run the task now
Async.RunSynchronously timerCount1
```

Let's see how we would do this same thing in a functional way:

```
// create the timer and the corresponding observable
let timerCount2, timerEventStream = createTimerAndObservable 500

// set up the transformations on the event stream
timerEventStream
|> Observable.scan (fun count _ -> count + 1) 0
|> Observable.subscribe (fun count -> printfn "timer ticked with count %i" count)

// run the task now
Async.RunSynchronously timerCount2
```

Here we see how you can build up layers of event transformations, just as you do with list transformations in LINQ.

The first transformation is `scan`, which accumulates state for each event. It is roughly equivalent to the `List.fold` function that we have seen used with lists. In this case, the accumulated state is just a counter.

And then, for each event, the count is printed out.

Note that in this functional approach, we didn't have any mutable state, and we didn't need to create any special classes.

Merging multiple event streams

For a final example, we'll look at merging multiple event streams.

Let's make a requirement based on the well-known "FizzBuzz" problem:

```
Create two timers, called '3' and '5'. The '3' timer ticks every 300ms and the '5' tim
er ticks
every 500ms.

Handle the events as follows:
a) for all events, print the id of the time and the time
b) when a tick is simultaneous with a previous tick, print 'FizzBuzz'
otherwise:
c) when the '3' timer ticks on its own, print 'Fizz'
d) when the '5' timer ticks on its own, print 'Buzz'
```

First let's create some code that both implementations can use.

We'll want a generic event type that captures the timer id and the time of the tick.

```
type FizzBuzzEvent = {label:int; time: DateTime}
```

And then we need a utility function to see if two events are simultaneous. We'll be generous and allow a time difference of up to 50ms.

```
let areSimultaneous (earlierEvent,laterEvent) =
    let {label=_;time=t1} = earlierEvent
    let {label=_;time=t2} = laterEvent
    t2.Subtract(t1).Milliseconds < 50
```

In the imperative design, we'll need to keep track of the previous event, so we can compare them. And we'll need special case code for the first time, when the previous event doesn't exist

```
type ImperativeFizzBuzzHandler() =

    let mutable previousEvent: FizzBuzzEvent option = None

    let printEvent thisEvent  =
      let {label=id; time=t} = thisEvent
      printf "[%i] %i.%03i " id t.Second t.Millisecond
      let simultaneous = previousEvent.IsSome && areSimultaneous (previousEvent.Value,
thisEvent)
      if simultaneous then printfn "FizzBuzz"
      elif id = 3 then printfn "Fizz"
      elif id = 5 then printfn "Buzz"

    member this.handleEvent3 eventArgs =
      let event = {label=3; time=DateTime.Now}
      printEvent event
      previousEvent <- Some event

    member this.handleEvent5 eventArgs =
      let event = {label=5; time=DateTime.Now}
      printEvent event
      previousEvent <- Some event
```

Now the code is beginning to get ugly fast! Already we have mutable state, complex conditional logic, and special cases, just for such a simple requirement.

Let's test it:

```
// create the class
let handler = new ImperativeFizzBuzzHandler()

// create the two timers and register the two handlers
let timer3 = createTimer 300 handler.handleEvent3
let timer5 = createTimer 500 handler.handleEvent5

// run the two timers at the same time
[timer3;timer5]
|> Async.Parallel
|> Async.RunSynchronously
```

It does work, but are you sure the code is not buggy? Are you likely to accidentally break something if you change it?

The problem with this imperative code is that it has a lot of noise that obscures the the requirements.

Can the functional version do better? Let's see!

First, we create *two* event streams, one for each timer:

```
let timer3, timerEventStream3 = createTimerAndObservable 300
let timer5, timerEventStream5 = createTimerAndObservable 500
```

Next, we convert each event on the "raw" event streams into our FizzBuzz event type:

```
// convert the time events into FizzBuzz events with the appropriate id
let eventStream3  =
   timerEventStream3
   |> Observable.map (fun _ -> {label=3; time=DateTime.Now})

let eventStream5  =
   timerEventStream5
   |> Observable.map (fun _ -> {label=5; time=DateTime.Now})
```

Now, to see if two events are simultaneous, we need to compare them from the two different streams somehow.

It's actually easier than it sounds, because we can:

- combine the two streams into a single stream:
- then create pairs of sequential events
- then test the pairs to see if they are simultaneous
- then split the input stream into two new output streams based on that test

Here's the actual code to do this:

```
// combine the two streams
let combinedStream =
    Observable.merge eventStream3 eventStream5

// make pairs of events
let pairwiseStream =
   combinedStream |> Observable.pairwise

// split the stream based on whether the pairs are simultaneous
let simultaneousStream, nonSimultaneousStream =
    pairwiseStream |> Observable.partition areSimultaneous
```

Finally, we can split the `nonSimultaneousStream` again, based on the event id:

```
// split the non-simultaneous stream based on the id
let fizzStream, buzzStream  =
    nonSimultaneousStream
    // convert pair of events to the first event
    |> Observable.map (fun (ev1,_) -> ev1)
    // split on whether the event id is three
    |> Observable.partition (fun {label=id} -> id=3)
```

Let's review so far. We have started with the two original event streams and from them created four new ones:

- `combinedStream` contains all the events
- `simultaneousStream` contains only the simultaneous events
- `fizzStream` contains only the non-simultaneous events with id=3
- `buzzStream` contains only the non-simultaneous events with id=5

Now all we need to do is attach behavior to each stream:

```
//print events from the combinedStream
combinedStream
|> Observable.subscribe (fun {label=id;time=t} ->
                              printf "[%i] %i.%03i " id t.Second t.Millisecond)

//print events from the simultaneous stream
simultaneousStream
|> Observable.subscribe (fun _ -> printfn "FizzBuzz")

//print events from the nonSimultaneous streams
fizzStream
|> Observable.subscribe (fun _ -> printfn "Fizz")

buzzStream
|> Observable.subscribe (fun _ -> printfn "Buzz")
```

Let's test it:

```
// run the two timers at the same time
[timer3;timer5]
|> Async.Parallel
|> Async.RunSynchronously
```

Here's all the code in one complete set:

```
// create the event streams and raw observables
let timer3, timerEventStream3 = createTimerAndObservable 300
let timer5, timerEventStream5 = createTimerAndObservable 500

// convert the time events into FizzBuzz events with the appropriate id
let eventStream3  = timerEventStream3
                    |> Observable.map (fun _ -> {label=3; time=DateTime.Now})
let eventStream5  = timerEventStream5
                    |> Observable.map (fun _ -> {label=5; time=DateTime.Now})

// combine the two streams
let combinedStream =
   Observable.merge eventStream3 eventStream5

// make pairs of events
let pairwiseStream =
   combinedStream |> Observable.pairwise

// split the stream based on whether the pairs are simultaneous
let simultaneousStream, nonSimultaneousStream =
   pairwiseStream |> Observable.partition areSimultaneous

// split the non-simultaneous stream based on the id
let fizzStream, buzzStream  =
    nonSimultaneousStream
    // convert pair of events to the first event
    |> Observable.map (fun (ev1,_) -> ev1)
    // split on whether the event id is three
    |> Observable.partition (fun {label=id} -> id=3)

//print events from the combinedStream
combinedStream
|> Observable.subscribe (fun {label=id;time=t} ->
                              printf "[%i] %i.%03i " id t.Second t.Millisecond)

//print events from the simultaneous stream
simultaneousStream
|> Observable.subscribe (fun _ -> printfn "FizzBuzz")

//print events from the nonSimultaneous streams
fizzStream
|> Observable.subscribe (fun _ -> printfn "Fizz")

buzzStream
|> Observable.subscribe (fun _ -> printfn "Buzz")

// run the two timers at the same time
[timer3;timer5]
|> Async.Parallel
|> Async.RunSynchronously
```

The code might seem a bit long winded, but this kind of incremental, step-wise approach is very clear and self-documenting.

Some of the benefits of this style are:

- I can see that it meets the requirements just by looking at it, without even running it. Not so with the imperative version.
- From a design point of view, each final "output" stream follows the single responsibility principle -- it only does one thing -- so it is very easy to associate behavior with it.
- This code has no conditionals, no mutable state, no edge cases. It would be easy to maintain or change, I hope.
- It is easy to debug. For example, I could easily "tap" the output of the `simultaneousStream` to see if it contains what I think it contains:

```
// debugging code
//simultaneousStream |> Observable.subscribe (fun e -> printfn "sim %A" e)
//nonSimultaneousStream |> Observable.subscribe (fun e -> printfn "non-sim %A" e)
```

This would be much harder in the imperative version.

Summary

Functional Reactive Programming (known as FRP) is a big topic, and we've only just touched on it here. I hope this introduction has given you a glimpse of the usefulness of this way of doing things.

If you want to learn more, see the documentation for the F# Observable module, which has the basic transformations used above. And there is also the Reactive Extensions (Rx) library which shipped as part of .NET 4. That contains many other additional transformations.

Completeness

In this final set of posts, we will look at some other aspects of F# under the theme of "completeness".

Programming languages coming from the academic world tend to focus on elegance and purity over real-world usefulness, while more mainstream business languages such as C# and Java are valued precisely because they are pragmatic; they can work in a wide array of situations and have extensive tools and libraries to meet almost every need. In other words, to be useful in the enterprise, a language needs to be *complete*, not just well-designed.

F# is unusual in that it successfully bridges both worlds. Although all the examples so far have focused on F# as an elegant functional language, it does support an object-oriented paradigm as well, and can integrate easily with other .NET languages and tools. As a result, F# is not a isolated island, but benefits from being part of the whole .NET ecosystem.

The other aspects that make F# "complete" are being an official .NET language (with all the support and documentation that that entails) and being designed to work in Visual Studio (which provides a nice editor with IntelliSense support, a debugger, and so on). These benefits should be obvious and won't be discussed here.

So, in this last section, we'll focus on two particular areas:

- **Seamless interoperation with .NET libraries**. Obviously, there can be a mismatch between the functional approach of F# and the imperative approach that is designed into the base libraries. We'll look at some of the features of F# that make this integration easier.
- **Full support for classes and other C# style code**. F# is designed as a hybrid functional/OO language, so it can do almost everything that C# can do as well. We'll have a quick tour of the syntax for these other features.

Seamless interoperation with .NET libraries

We have already seen many examples of F#'s use with the .NET libraries, such as using `System.Net.WebRequest` and `System.Text.RegularExpressions`. And the integration was indeed seamless.

For more complex requirements, F# natively supports .NET classes, interfaces, and structures, so the interop is still very straightforward. For example, you can write an `ISomething` interface in C# and have the implementation be done in F#.

But not only can F# call into existing .NET code, it can also expose almost any .NET API back to other languages. For example, you can write classes and methods in F# and expose them to C#, VB or COM. You can even do the above example backwards -- define an `ISomething` interface in F# and have the implementation be done in C#! The benefit of all this is that you don't have to discard any of your existing code base; you can start using F# for some things while retaining C# or VB for others, and pick the best tool for the job.

In addition to the tight integration though, there are a number of nice features in F# that often make working with .NET libraries more convenient than C# in some ways. Here are some of my favorites:

- You can use `TryParse` and `TryGetValue` without passing an "out" parameter.
- You can resolve method overloads by using argument names, which also helps with type inference.
- You can use "active patterns" to convert .NET APIs into more friendly code.
- You can dynamically create objects from an interface such as `IDisposable` without creating a concrete class.
- You can mix and match "pure" F# objects with existing .NET APIs

TryParse and TryGetValue

The `TryParse` and `TryGetValue` functions for values and dictionaries are frequently used to avoid extra exception handling. But the C# syntax is a bit clunky. Using them from F# is more elegant because F# will automatically convert the function into a tuple where the first element is the function return value and the second is the "out" parameter.

```
//using an Int32
let (i1success,i1) = System.Int32.TryParse("123");
if i1success then printfn "parsed as %i" i1 else printfn "parse failed"

let (i2success,i2) = System.Int32.TryParse("hello");
if i2success then printfn "parsed as %i" i2 else printfn "parse failed"

//using a DateTime
let (d1success,d1) = System.DateTime.TryParse("1/1/1980");
let (d2success,d2) = System.DateTime.TryParse("hello");

//using a dictionary
let dict = new System.Collections.Generic.Dictionary<string,string>();
dict.Add("a","hello")
let (e1success,e1) = dict.TryGetValue("a");
let (e2success,e2) = dict.TryGetValue("b");
```

Named arguments to help type inference

In C# (and .NET in general), you can have overloaded methods with many different parameters. F# can have trouble with this. For example, here is an attempt to create a `StreamReader` :

```
let createReader fileName = new System.IO.StreamReader(fileName)
// error FS0041: A unique overload for method 'StreamReader'
//               could not be determined
```

The problem is that F# does not know if the argument is supposed to be a string or a stream. You could explicitly specify the type of the argument, but that is not the F# way!

Instead, a nice workaround is enabled by the fact that in F#, when calling methods in .NET libraries, you can specify named arguments.

```
let createReader2 fileName = new System.IO.StreamReader(path=fileName)
```

In many cases, such as the one above, just using the argument name is enough to resolve the type issue. And using explicit argument names can often help to make the code more legible anyway.

Active patterns for .NET functions

There are many situations where you want to use pattern matching against .NET types, but the native libraries do not support this. Earlier, we briefly touched on the F# feature called "active patterns" which allows you to dynamically create choices to match on. This can be very for useful .NET integration.

A common case is that a .NET library class has a number of mutually exclusive `isSomething`, `isSomethingElse` methods, which have to be tested with horrible looking cascading if-else statements. Active patterns can hide all the ugly testing, letting the rest of your code use a more natural approach.

For example, here's the code to test for various `isXXX` methods for `System.Char`.

```
let (|Digit|Letter|Whitespace|Other|) ch =
   if System.Char.IsDigit(ch) then Digit
   else if System.Char.IsLetter(ch) then Letter
   else if System.Char.IsWhiteSpace(ch) then Whitespace
   else Other
```

Once the choices are defined, the normal code can be straightforward:

```
let printChar ch =
  match ch with
  | Digit -> printfn "%c is a Digit" ch
  | Letter -> printfn "%c is a Letter" ch
  | Whitespace -> printfn "%c is a Whitespace" ch
  | _ -> printfn "%c is something else" ch

// print a list
['a';'b';'1';' ';'-';'c'] |> List.iter printChar
```

Another common case is when you have to parse text or error codes to determine the type of an exception or result. Here's an example that uses an active pattern to parse the error number associated with `SqlExceptions`, making them more palatable.

First, set up the active pattern matching on the error number:

```
open System.Data.SqlClient

let (|ConstraintException|ForeignKeyException|Other|) (ex:SqlException) =
   if ex.Number = 2601 then ConstraintException
   else if ex.Number = 2627 then ConstraintException
   else if ex.Number = 547 then ForeignKeyException
   else Other
```

Now we can use these patterns when processing SQL commands:

```
let executeNonQuery (sqlCommmand:SqlCommand) =
    try
       let result = sqlCommmand.ExecuteNonQuery()
       // handle success
    with
    | :?SqlException as sqlException -> // if a SqlException
        match sqlException with         // nice pattern matching
        | ConstraintException  -> // handle constraint error
        | ForeignKeyException  -> // handle FK error
        | _ -> reraise()          // don't handle any other cases
    // all non SqlExceptions are thrown normally
```

Creating objects directly from an interface

F# has another useful feature called "object expressions". This is the ability to directly create objects from an interface or abstract class without having to define a concrete class first.

In the example below, we create some objects that implement `IDisposable` using a `makeResource` helper function.

```
// create a new object that implements IDisposable
let makeResource name =
   { new System.IDisposable
     with member this.Dispose() = printfn "%s disposed" name }

let useAndDisposeResources =
    use r1 = makeResource "first resource"
    printfn "using first resource"
    for i in [1..3] do
        let resourceName = sprintf "\tinner resource %d" i
        use temp = makeResource resourceName
        printfn "\tdo something with %s" resourceName
    use r2 = makeResource "second resource"
    printfn "using second resource"
    printfn "done."
```

The example also demonstrates how the " `use` " keyword automatically disposes a resource when it goes out of scope. Here is the output:

```
using first resource
    do something with     inner resource 1
    inner resource 1 disposed
    do something with     inner resource 2
    inner resource 2 disposed
    do something with     inner resource 3
    inner resource 3 disposed
using second resource
done.
second resource disposed
first resource disposed
```

Mixing .NET interfaces with pure F# types

The ability to create instances of an interface on the fly means that it is easy to mix and match interfaces from existing APIs with pure F# types.

For example, say that you have a preexisting API which uses the `IAnimal` interface, as shown below.

```
type IAnimal =
   abstract member MakeNoise : unit -> string

let showTheNoiseAnAnimalMakes (animal:IAnimal) =
   animal.MakeNoise() |> printfn "Making noise %s"
```

But we want to have all the benefits of pattern matching, etc, so we have created pure F# types for cats and dogs instead of classes.

```
type Cat = Felix | Socks
type Dog = Butch | Lassie
```

But using this pure F# approach means that that we cannot pass the cats and dogs to the `showTheNoiseAnAnimalMakes` function directly.

However, we don't have to create new sets of concrete classes just to implement `IAnimal`. Instead, we can dynamically create the `IAnimal` interface by extending the pure F# types.

```
// now mixin the interface with the F# types
type Cat with
   member this.AsAnimal =
        { new IAnimal
          with member a.MakeNoise() = "Meow" }

type Dog with
   member this.AsAnimal =
        { new IAnimal
          with member a.MakeNoise() = "Woof" }
```

Here is some test code:

```
let dog = Lassie
showTheNoiseAnAnimalMakes (dog.AsAnimal)

let cat = Felix
showTheNoiseAnAnimalMakes (cat.AsAnimal)
```

This approach gives us the best of both worlds. Pure F# types internally, but the ability to convert them into interfaces as needed to interface with libraries.

Using reflection to examine F# types

F# gets the benefit of the .NET reflection system, which means that you can do all sorts of interesting things that are not directly available to you using the syntax of the language itself. The `Microsoft.FSharp.Reflection` namespace has a number of functions that are designed to help specifically with F# types.

For example, here is a way to print out the fields in a record type, and the choices in a union type.

```
open System.Reflection
open Microsoft.FSharp.Reflection

// create a record type...
type Account = {Id: int; Name: string}

// ... and show the fields
let fields =
    FSharpType.GetRecordFields(typeof<Account>)
    |> Array.map (fun propInfo -> propInfo.Name, propInfo.PropertyType.Name)

// create a union type...
type Choices = | A of int | B of string

// ... and show the choices
let choices =
    FSharpType.GetUnionCases(typeof<Choices>)
    |> Array.map (fun choiceInfo -> choiceInfo.Name)
```

Anything C# can do...

As should be apparent, you should generally try to prefer functional-style code over object-oriented code in F#, but in some situations, you may need all the features of a fully fledged OO language ? classes, inheritance, virtual methods, etc.

So just to conclude this section, here is a whirlwind tour of the F# versions of these features.

Some of these will be dealt with in much more depth in a later series on .NET integration. But I won't cover some of the more obscure ones, as you can read about them in the MSDN documentation if you ever need them.

Classes and interfaces

First, here are some examples of an interface, an abstract class, and a concrete class that inherits from the abstract class.

```
// interface
type IEnumerator<'a> =
    abstract member Current : 'a
    abstract MoveNext : unit -> bool

// abstract base class with virtual methods
[<AbstractClass>]
type Shape() =
    //readonly properties
    abstract member Width : int with get
    abstract member Height : int with get
    //non-virtual method
    member this.BoundingArea = this.Height * this.Width
    //virtual method with base implementation
    abstract member Print : unit -> unit
    default this.Print () = printfn "I'm a shape"

// concrete class that inherits from base class and overrides
type Rectangle(x:int, y:int) =
    inherit Shape()
    override this.Width = x
    override this.Height = y
    override this.Print ()  = printfn "I'm a Rectangle"

//test
let r = Rectangle(2,3)
printfn "The width is %i" r.Width
printfn "The area is %i" r.BoundingArea
r.Print()
```

Classes can have multiple constructors, mutable properties, and so on.

```
type Circle(rad:int) =
    inherit Shape()

    //mutable field
    let mutable radius = rad

    //property overrides
    override this.Width = radius * 2
    override this.Height = radius * 2

    //alternate constructor with default radius
    new() = Circle(10)

    //property with get and set
    member this.Radius
         with get() = radius
         and set(value) = radius <- value

// test constructors
let c1 = Circle()   // parameterless ctor
printfn "The width is %i" c1.Width
let c2 = Circle(2)  // main ctor
printfn "The width is %i" c2.Width

// test mutable property
c2.Radius <- 3
printfn "The width is %i" c2.Width
```

Generics

F# supports generics and all the associated constraints.

```
// standard generics
type KeyValuePair<'a,'b>(key:'a, value: 'b) =
    member this.Key = key
    member this.Value = value

// generics with constraints
type Container<'a,'b
    when 'a : equality
    and 'b :> System.Collections.ICollection>
    (name:'a, values:'b) =
    member this.Name = name
    member this.Values = values
```

Structs

F# supports not just classes, but the .NET struct types as well, which can help to boost performance in certain cases.

```
type Point2D =
   struct
      val X: float
      val Y: float
      new(x: float, y: float) = { X = x; Y = y }
   end

//test
let p = Point2D()  // zero initialized
let p2 = Point2D(2.0,3.0)  // explicitly initialized
```

Exceptions

F# can create exception classes, raise them and catch them.

```
// create a new Exception class
exception MyError of string

try
    let e = MyError("Oops!")
    raise e
with
    | MyError msg ->
        printfn "The exception error was %s" msg
    | _ ->
        printfn "Some other exception"
```

Extension methods

Just as in C#, F# can extend existing classes with extension methods.

```
type System.String with
    member this.StartsWithA = this.StartsWith "A"

//test
let s = "Alice"
printfn "'%s' starts with an 'A' = %A" s s.StartsWithA

type System.Int32 with
    member this.IsEven = this % 2 = 0

//test
let i = 20
if i.IsEven then printfn "'%i' is even" i
```

Parameter arrays

Just like C#'s variable length "params" keyword, this allows a variable length list of arguments to be converted to a single array parameter.

```
open System
type MyConsole() =
    member this.WriteLine([<ParamArray>] args: Object[]) =
        for arg in args do
            printfn "%A" arg

let cons = new MyConsole()
cons.WriteLine("abc", 42, 3.14, true)
```

Events

F# classes can have events, and the events can be triggered and responded to.

```
type MyButton() =
    let clickEvent = new Event<_>()

    [<CLIEvent>]
    member this.OnClick = clickEvent.Publish

    member this.TestEvent(arg) =
        clickEvent.Trigger(this, arg)

// test
let myButton = new MyButton()
myButton.OnClick.Add(fun (sender, arg) ->
        printfn "Click event with arg=%O" arg)

myButton.TestEvent("Hello World!")
```

Delegates

F# can do delegates.

```
// delegates
type MyDelegate = delegate of int -> int
let f = MyDelegate (fun x -> x * x)
let result = f.Invoke(5)
```

Enums

F# supports CLI enums types, which look similar to the "union" types, but are actually different behind the scenes.

```
// enums
type Color = | Red=1 | Green=2 | Blue=3

let color1  = Color.Red    // simple assignment
let color2:Color = enum 2  // cast from int
// created from parsing a string
let color3 = System.Enum.Parse(typeof<Color>,"Green") :?> Color // :?> is a downcast

[<System.FlagsAttribute>]
type FileAccess = | Read=1 | Write=2 | Execute=4
let fileaccess = FileAccess.Read ||| FileAccess.Write
```

Working with the standard user interface

Finally, F# can work with the WinForms and WPF user interface libraries, just like C#.

Here is a trivial example of opening a form and handling a click event.

```
open System.Windows.Forms

let form = new Form(Width= 400, Height = 300, Visible = true, Text = "Hello World")
form.TopMost <- true
form.Click.Add (fun args-> printfn "the form was clicked")
form.Show()
```

Why use F#: Conclusion

This completes the tour of F# features. I hope that the examples have given you some appreciation of the power of F# and functional programming. If you have any comments on the series as a whole, please leave them at the bottom of this page.

In later series I hope to go deeper into data structures, pattern matching, list processing, asynchronous and parallel programming, and much more.

But before those, I recommend you read the "thinking functionally" series, which will help you understand functional programming at a deeper level.

This series of posts will introduce you to the fundamentals of functional programming -- what does it really mean to "program functionally", and how this approach differs from object oriented or imperative programming.

- Thinking Functionally: Introduction. A look at the basics of functional programming.
- Mathematical functions. The impetus behind functional programming.
- Function Values and Simple Values. Binding not assignment.
- How types work with functions. Understanding the type notation.
- Currying. Breaking multi-parameter functions into smaller one-parameter functions.
- Partial application. Baking-in some of the parameters of a function.
- Function associativity and composition. Building new functions from existing ones.
- Defining functions. Lambdas and more.
- Function signatures. A function signature can give you some idea of what it does.
- Organizing functions. Nested functions and modules.
- Attaching functions to types. Creating methods the F# way.
- Worked example: A stack based calculator. Using combinators to build functionality.

Thinking Functionally: Introduction

Now that you have seen some of the power of F# in the "why use F#" series, we're going to step back and look at the fundamentals of functional programming -- what does it really mean to "program functionally", and how this approach is different from object oriented or imperative programming.

Changing the way you think

It is important to understand that functional programming is not just a stylistic difference; it is a completely different way of thinking about programming, in the way that truly object-oriented programming (in Smalltalk say) is also a different way of thinking from a traditional imperative language such as C.

F# does allow non-functional styles, and it is tempting to retain the habits you already are familiar with. You could just use F# in a non-functional way without really changing your mindset, and not realize what you are missing. To get the most out of F#, and to be fluent and comfortable with functional programming in general, it is critical that you think functionally, not imperatively. And that is the goal of this series: to help you understand functional programming in a deep way, and help to change the way you think.

This will be a quite abstract series, although I will use lots of short code examples to demonstrate the points. We will cover the following points:

- **Mathematical functions**. The first post introduces the mathematical ideas behind functional languages, and the benefits that come from this approach.
- **Functions and values**. The next post introduces functions and values, showing how "values" are different from variables, and why there are similarities between function and simple values.
- **Types**. Then we move on to the basic types that work with functions: primitive types such as string and int; the unit type, function types, and generic types.
- **Functions with multiple parameters**. Next, I explain the concepts of "currying" and "partial application". This is where your brain can start to hurt, if you are coming from an imperative background!
- **Defining functions**. Then some posts devoted to the many different ways to define and combine functions.
- **Function signatures.** Then a important post on the critical topic of function signatures: what they mean and how to use them as an aid to understanding.
- **Organizing functions.** Once you know how to create functions, how can you organize them to make them available to the rest of your code?

Mathematical functions

The impetus behind functional programming comes from mathematics. Mathematical functions have a number of very nice features that functional languages try to emulate in the real world.

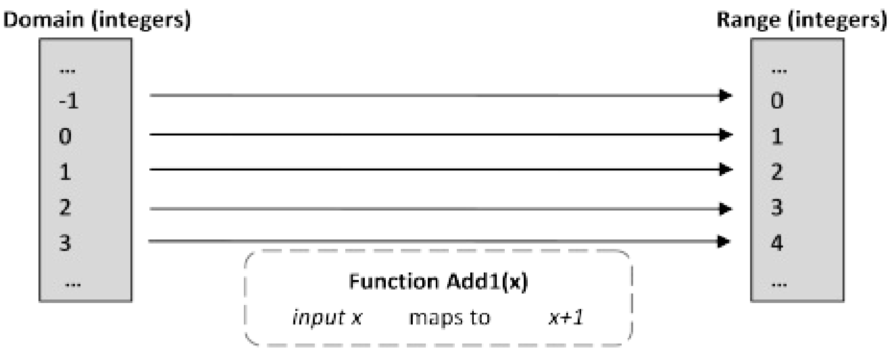
So first, let's start with a mathematical function that adds 1 to a number.

```
Add1(x) = x+1
```

What does this really mean? Well it seems pretty straightforward. It means that there is an operation that starts with a number, and adds one to it.

Let's introduce some terminology:

- The set of values that can be used as input to the function is called the *domain*. In this case, it could be the set of real numbers, but to make life simpler for now, let's restrict it to integers only.
- The set of possible output values from the function is called the *range* (technically, the image on the codomain). In this case, it is also the set of integers.
- The function is said to *map* the domain to the range.



Here's how the definition would look in F#

```
let add1 x = x + 1
```

If you type that into the F# interactive window (don't forget the double semicolons) you will see the result (the "signature" of the function):

```
val add1 : int -> int
```

Let's look at that output in detail:

- The overall meaning is "the function `add1` maps integers (the domain) onto integers (the range)".
- " `add1` " is defined as a "val", short for "value". Hmmm? what does that mean? We'll discuss values shortly.
- The arrow notation " `->` " is used to show the domain and range. In this case, the domain is the `int` type, and the range is also the `int` type.

Also note that the type was not specified, yet the F# compiler guessed that the function was working with ints. (Can this be tweaked? Yes, as we'll see shortly).

Key properties of mathematical functions

Mathematical functions have some properties that are very different from the kinds of functions you are used to in procedural programming.

- A function always gives the same output value for a given input value
- A function has no side effects.

These properties provide some very powerful benefits, and so functional programming languages try to enforce these properties in their design as well. Let's look at each of them in turn.

Mathematical functions always give the same output for a given input

In imperative programming, we think that functions "do" something or "calculate" something. A mathematical function does not do any calculation -- it is purely a mapping from input to output. In fact, another way to think of defining a function is simply as the set of all the mappings. For example, in a very crude way we could define the " `add1` " function (in C#) as

```
int add1(int input)
{
   switch (input)
   {
   case 0: return 1;
   case 1: return 2;
   case 2: return 3;
   case 3: return 4;
   etc ad infinitum
   }
}
```

Obviously, we can't have a case for every possible integer, but the principle is the same. You can see that absolutely no calculation is being done at all, just a lookup.

Mathematical functions are free from side effects

In a mathematical function, the input and the output are logically two different things, both of which are predefined. The function does not change the input or the output -- it just maps a pre-existing input value from the domain to a pre-existing output value in the range.

In other words, evaluating the function *cannot possibly have any effect on the input, or anything else for that matter*. Remember, evaluating the function is not actually calculating or manipulating anything; it is just a glorified lookup.

This "immutability" of the values is subtle but very important. If I am doing mathematics, I do not expect the numbers to change underneath me when I add them! For example, if I have:

```
x = 5
y = x+1
```

I would not expect x to be changed by the adding of one to it. I would expect to get back a different number (y) and x would be left untouched. In the world of mathematics, the integers already exist as an unchangeable set, and the "add1" function simply defines a relationship between them.

The power of pure functions

The kinds of functions which have repeatable results and no side effects are called "pure functions", and you can do some interesting things with them:

- They are trivially parallelizable. I could take all the integers from 1 to 1000, say, and given 1000 different CPUs, I could get each CPU to execute the " `add1` " function for the corresponding integer at the same time, safe in the knowledge that there was no need for any interaction between them. No locks, mutexes, semaphores, etc., needed.
- I can use a function lazily, only evaluating it when I need the output. I can be sure that the answer will be the same whether I evaluate it now or later.
- I only ever need to evaluate a function once for a certain input, and I can then cache the result, because I know that the same input always gives the same output.
- If I have a number of pure functions, I can evaluate them in any order I like. Again, it can't make any difference to the final result.

So you can see that if we can create pure functions in a programming language, we immediately gain a lot of powerful techniques. And indeed you can do all these things in F#:

- You have already seen an example of parallelism in the "why use F#?" series.
- Evaluating functions lazily will be discussed in the "optimization" series.
- Caching the results of functions is called "memoization" and will also be discussed in the "optimization" series.
- Not caring about the order of evaluation makes concurrent programming much easier, and doesn't introduce bugs when functions are reordered or refactored.

"Unhelpful" properties of mathematical functions

Mathematical functions also have some properties that seem not to be very helpful when used in programming.

- The input and output values are immutable
- A function always has exactly one input and one output

These properties are mirrored in the design of functional programming languages too. Let's look at each of these in turn.

The input and output values are immutable

Immutable values seem like a nice idea in theory, but how can you actually get any work done if you can't assign to variables in a traditional way?

I can assure you that this is not as much as a problem as you might think. As you work through this series, you'll see how this works in practice.

Mathematical functions always have exactly one input and one output

As you can see from the diagrams, there is always exactly one input and one output for a mathematical function. This is true for functional programming languages as well, although it may not be obvious when you first use them.

That seems like a big annoyance. How can you do useful things without having functions with two (or more) parameters?

Well, it turns there is a way to do it, and what's more, it is completely transparent to you in F#. It is called "currying" and it deserves its own post, which is coming up soon.

In fact, as you will later discover, these two "unhelpful" properties will turn out to be incredibly useful and a key part of what makes functional programming so powerful.

Function Values and Simple Values

Let's look at the simple function again

```
let add1 x = x + 1
```

What does the "x" mean here? It means:

1. Accept some value from the input domain.
2. Use the name "x" to represent that value so that we can refer to it later.

This process of using a name to represent a value is called "binding". The name "x" is "bound" to the input value.

So if we evaluate the function with the input 5 say, what is happening is that everywhere we see "x" in the original definition, we replace it with "5", sort of like search and replace in a word processor.

```
let add1 x = x + 1
add1 5
// replace "x" with "5"
// add1 5 = 5 + 1 = 6
// result is 6
```

It is important to understand that this is not assignment. "x" is not a "slot" or variable that is assigned to the value and can be assigned to another value later on. It is a onetime association of the name "x" with the value. The value is one of the predefined integers, and cannot change. And so, once bound, x cannot change either; once associated with a value, always associated with a value.

This concept is a critical part of thinking functionally: *there are no "variables", only values*.

Function values

If you think about this a bit more, you will see that the name " `add1` " itself is just a binding to "the function that adds one to its input". The function itself is independent of the name it is bound to.

When you type `let add1 x = x + 1` you are telling the F# compiler "every time you see the name " `add1` ", replace it with the function that adds 1 to its input". " `add1` " is called a **function value**.

To see that the function is independent of its name, try:

```
let add1 x = x + 1
let plus1 = add1
add1 5
plus1 5
```

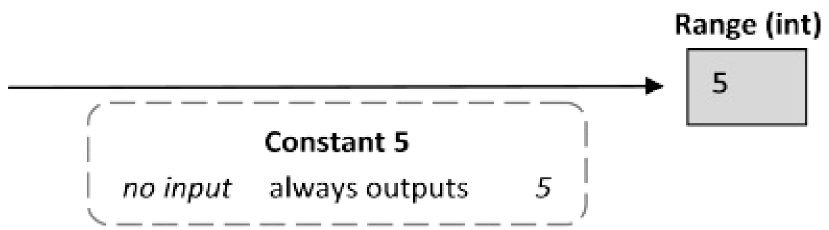
You can see that " `add1` " and " `plus1` " are two names that refer ("bound to") to the same function.

You can always identify a function value because its signature has the standard form `domain -> range`. Here is a generic function value signature:

```
val functionName : domain -> range
```

Simple values

Imagine an operation that always returned the integer 5 and didn't have any input.



This would be a "constant" operation.

How would we write this in F#? We want to tell the F# compiler "every time you see the name `c`, replace it with 5". Here's how:

```
let c = 5
```

which when evaluated, returns:

```
val c : int = 5
```

There is no mapping arrow this time, just a single int. What's new is an equals sign with the actual value printed after it. The F# compiler knows that this binding has a known value which it will always return, namely the value 5.

In other words, we've just defined a constant, or in F# terms, a simple value.

You can always tell a simple value from a function value because all simple values have a signature that looks like:

```
val aName: type = constant     // Note that there is no arrow
```

Simple values vs. function values

It is important to understand that in F#, unlike languages such as C#, there is very little difference between simple values and function values. They are both values which can be bound to names (using the same keyword `let`) and then passed around. And in fact, one of the key aspects of thinking functionally is exactly that: *functions are values that can be passed around as inputs to other functions*, as we will soon see.

Note that there is a subtle difference between a simple value and a function value. A function always has a domain and range and must be "applied" to an argument to get a result. A simple value does not need to be evaluated after being bound. Using the example above, if we wanted to define a "constant function" that returns five we would have to use

```
let c = fun()->5
// or
let c() = 5
```

The signature for these functions is:

```
val c : unit -> int
```

instead of:

```
val c : int = 5
```

More on unit, function syntax and anonymous functions later.

"Values" vs. "Objects"

In a functional programming language like F#, most things are called "values". In an object-oriented language like C#, most things are called "objects". So what is the difference between a "value" and an "object"?

A value, as we have seen above, is just a member of a domain. The domain of ints, the domain of strings, the domain of functions that map ints to strings, and so on. In principle, values are immutable. And values do not have any behavior attached them.

An object, in a standard definition, is an encapsulation of a data structure with its associated behavior (methods). In general, objects are expected to have state (that is, be mutable), and all operations that change the internal state must be provided by the object itself (via "dot" notation).

In F#, even the primitive values have some object-like behavior. For example, you can dot into a string to get its length:

```
"abc".Length
```

But, in general, we will avoid using "object" for standard values in F#, reserving it to refer to instances of true classes, or other values that expose member methods.

Naming Values

Standard naming rules are used for value and function names, basically, any alphanumeric string, including underscores. There are a couple of extras:

You can put an apostrophe anywhere in a name, except the first character. So:

```
A'b'c     begin'  // valid names
```

The final tick is often used to signal some sort of "variant" version of a value:

```
let f = x
let f' = derivative f
let f'' = derivative f'
```

or define variants of existing keywords

```
let if' b t f = if b then t else f
```

You can also put double backticks around any string to make a valid identifier.

```
``this is a name``  ``123``    //valid names
```

You might want to use the double backtick trick sometimes:

- When you want to use an identifier that is the same as a keyword

```
let ``begin`` = "begin"
```

- When trying to use natural language for business rules, unit tests, or BDD style executable specifications a la Cucumber.

```
let ``is first time customer?`` = true
let ``add gift to order`` = ()
if ``is first time customer?`` then ``add gift to order``

// Unit test
let [<Test>] ``When input is 2 then expect square is 4``=
   // code here

// BDD clause
let [<Given>] ``I have (.*) N products in my cart`` (n:int) =
   // code here
```

Unlike C#, the naming convention for F# is that functions and values start with lowercase letters rather than uppercase (`camelCase` rather than `PascalCase`) unless designed for exposure to other .NET languages. Types and modules use uppercase however.

How types work with functions

Now that we have some understanding of functions, we'll look at how types work with functions, both as domains and ranges. This is just an overview; the series "understanding F# types" will cover types in detail.

First, we need to understand the type notation a bit more. We've seen that the arrow notation " `->` " is used to show the domain and range. So that a function signature always looks like:

```
val functionName : domain -> range
```

Here are some example functions:

```
let intToString x = sprintf "x is %i" x  // format int to string
let stringToInt x = System.Int32.Parse(x)
```

If you evaluate that in the F# interactive window, you will see the signatures:

```
val intToString : int -> string
val stringToInt : string -> int
```

This means:

- `intToString` has a domain of `int` which it maps onto the range `string` .
- `stringToInt` has a domain of `string` which it maps onto the range `int` .

Primitive types

The possible primitive types are what you would expect: string, int, float, bool, char, byte, etc., plus many more derived from the .NET type system.

Here are some more examples of functions using primitive types:

```
let intToFloat x = float x // "float" fn. converts ints to floats
let intToBool x = (x = 2)  // true if x equals 2
let stringToString x = x + " world"
```

and their signatures are:

```
val intToFloat : int -> float
val intToBool : int -> bool
val stringToString : string -> string
```

Type annotations

In the previous examples, the F# compiler correctly determined the types of the parameters and results. But this is not always the case. If you try the following code, you will get a compiler error:

```
let stringLength x = x.Length
   => error FS0072: Lookup on object of indeterminate type
```

The compiler does not know what type "x" is, and therefore does not know if "Length" is a valid method. In most cases, this can be fixed by giving the F# compiler a "type annotation" so that it knows which type to use. In the corrected version below, we indicate that the type of "x" is a string.

```
let stringLength (x:string) = x.Length
```

The parens around the `x:string` param are important. If they are missing, the compiler thinks that the return value is a string! That is, an "open" colon is used to indicate the type of the return value, as you can see in the example below.

```
let stringLengthAsInt (x:string) :int = x.Length
```

We're indicating that the x param is a string and the return value is an int.

Function types as parameters

A function that takes other functions as parameters, or returns a function, is called a **higher-order function** (sometimes abbreviated as HOF). They are used as a way of abstracting out common behavior. These kinds of functions are extremely common in F#; most of the standard libraries use them.

Consider a function `evalWith5ThenAdd2`, which takes a function as a parameter, then evaluates the function with the value 5, and adds 2 to the result:

```
let evalWith5ThenAdd2 fn = fn 5 + 2     // same as fn(5) + 2
```

The signature of this function looks like this:

```
val evalWith5ThenAdd2 : (int -> int) -> int
```

You can see that the domain is `(int->int)` and the range is `int`. What does that mean? It means that the input parameter is not a simple value, but a function, and what's more is restricted only to functions that map `ints` to `ints`. The output is not a function, just an int.

Let's try it:

```
let add1 x = x + 1      // define a function of type (int -> int)
evalWith5ThenAdd2 add1  // test it
```

gives:

```
val add1 : int -> int
val it : int = 8
```

"`add1`" is a function that maps ints to ints, as we can see from its signature. So it is a valid parameter for the `evalWith5ThenAdd2` function. And the result is 8.

By the way, the special word "`it`" is used for the last thing that was evaluated; in this case the result we want. It's not a keyword, just a convention.

Here's another one:

```
let times3 x = x * 3      // a function of type (int -> int)
evalWith5ThenAdd2 times3  // test it
```

gives:

```
val times3 : int -> int
val it : int = 17
```

"`times3`" is also a function that maps ints to ints, as we can see from its signature. So it is also a valid parameter for the `evalWith5ThenAdd2` function. And the result is 17.

Note that the input is sensitive to the types. If our input function uses `floats` rather than `ints`, it will not work. For example, if we have:

```
let times3float x = x * 3.0  // a function of type (float->float)
evalWith5ThenAdd2 times3float
```

Evaluating this will give an error:

```
error FS0001: Type mismatch. Expecting a int -> int but
              given a float -> float
```

meaning that the input function should have been an `int->int` function.

Functions as output

A function value can also be the output of a function. For example, the following function will generate an "adder" function that adds using the input value.

```
let adderGenerator numberToAdd = (+) numberToAdd
```

The signature is:

```
val adderGenerator : int -> (int -> int)
```

which means that the generator takes an `int`, and creates a function (the "adder") that maps `ints` to `ints`. Let's see how it works:

```
let add1 = adderGenerator 1
let add2 = adderGenerator 2
```

This creates two adder functions. The first generated function adds 1 to its input, and the second adds 2. Note that the signatures are just as we would expect them to be.

```
val add1 : (int -> int)
val add2 : (int -> int)
```

And we can now use these generated functions in the normal way. They are indistinguishable from functions defined explicitly

```
add1 5    // val it : int = 6
add2 5    // val it : int = 7
```

Using type annotations to constrain function types

In the first example, we had the function:

```
let evalWith5ThenAdd2 fn = fn 5 +2
    => val evalWith5ThenAdd2 : (int -> int) -> int
```

In this case F# could deduce that " `fn` " mapped `ints` to `ints`, so its signature would be `int->int`

But what is the signature of "fn" in this following case?

```
let evalWith5 fn = fn 5
```

Obviously, " `fn` " is some kind of function that takes an int, but what does it return? The compiler can't tell. If you do want to specify the type of the function, you can add a type annotation for function parameters in the same way as for a primitive type.

```
let evalWith5AsInt (fn:int->int) = fn 5
let evalWith5AsFloat (fn:int->float) = fn 5
```

Alternatively, you could also specify the return type instead.

```
let evalWith5AsString fn :string = fn 5
```

Because the main function returns a string, the " `fn` " function is also constrained to return a string, so no explicit typing is required for "fn".

The "unit" type

When programming, we sometimes want a function to do something without returning a value. Consider the function " `printInt` ", defined below. The function doesn't actually return anything. It just prints a string to the console as a side effect.

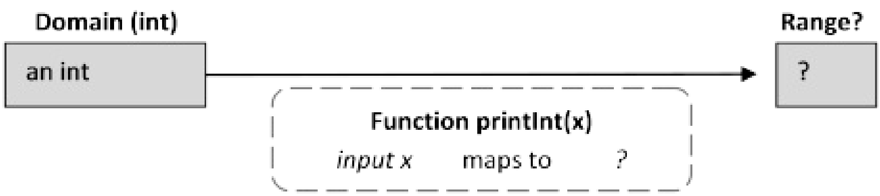
```
let printInt x = printf "x is %i" x        // print to console
```

So what is the signature for this function?

```
val printInt : int -> unit
```

What is this " `unit` "?

Well, even if a function returns no output, it still needs a range. There are no "void" functions in mathematics-land. Every function must have some output, because a function is a mapping, and a mapping has to have something to map to!



So in F#, functions like this return a special range called " `unit` ". This range has exactly one value in it, called " `()` ". You can think of `unit` and `()` as somewhat like "void" (the type) and "null" (the value) in C#. But unlike void/null, `unit` is a real type and `()` is a real value. To see this, evaluate:

```
let whatIsThis = ()
```

and you will see the signature:

```
val whatIsThis : unit = ()
```

Which means that the value " `whatIsThis` " is of type `unit` and has been bound to the value `()`

So, going back to the signature of " `printInt` ", we can now understand it:

```
val printInt : int -> unit
```

This signature says: `printInt` has a domain of `int` which it maps onto nothing that we care about.

Parameterless functions

Now that we understand unit, can we predict its appearance in other contexts? For example, let's try to create a reusable "hello world" function. Since there is no input and no output, we would expect it to have a signature `unit -> unit`. Let's see:

```
let printHello = printf "hello world"        // print to console
```

The result is:

```
hello world
val printHello : unit = ()
```

Not quite what we expected. "Hello world" is printed immediately and the result is not a function, but a simple value of type unit. As we saw earlier, we can tell that this is a simple value because it has a signature of the form:

```
val aName: type = constant
```

So in this case, we see that `printHello` is actually a *simple value* with the value `()`. It's not a function that we can call again.

Why the difference between `printInt` and `printHello`? In the `printInt` case, the value could not be determined until we knew the value of the x parameter, so the definition was of a function. In the `printHello` case, there were no parameters, so the right hand side could be determined immediately. Which it was, returning the `()` value, with the side effect of printing to the console.

We can create a true reusable function that is parameterless by forcing the definition to have a unit argument, like this:

```
let printHelloFn () = printf "hello world"    // print to console
```

The signature is now:

```
val printHelloFn : unit -> unit
```

and to call it, we have to pass the `()` value as a parameter, like so:

```
printHelloFn ()
```

Forcing unit types with the ignore function

In some cases the compiler requires a unit type and will complain. For example, both of the following will be compiler errors:

```
do 1+1     // => FS0020: This expression should have type 'unit'

let something =
  2+2      // => FS0020: This expression should have type 'unit'
  "hello"
```

To help in these situations, there is a special function `ignore` that takes anything and returns the unit type. The correct version of this code would be:

```
do (1+1 |> ignore)  // ok

let something =
  2+2 |> ignore     // ok
  "hello"
```

Generic types

In many cases, the type of the function parameter can be any type, so we need a way to indicate this. F# uses the .NET generic type system for this situation.

For example, the following function converts the parameter to a string and appends some text:

```
let onAStick x = x.ToString() + " on a stick"
```

It doesn't matter what type the parameter is, as all objects understand `ToString()`.

The signature is:

```
val onAStick : 'a -> string
```

What is this type called `'a`? That is F#'s way of indicating a generic type that is not known at compile time. The apostrophe in front of the "a" means that the type is generic. The signature for the C# equivalent of this would be:

```
string onAStick<a>();

//or more idiomatically
string OnAStick<TObject>();   // F#'s use of 'a is like
                              // C#'s "TObject" convention
```

Note that the F# function is still strongly typed with a generic type. It does *not* take a parameter of type `Object`. This strong typing is desirable so that when functions are composed together, type safety is still maintained.

Here's the same function being used with an int, a float and a string

```
onAStick 22
onAStick 3.14159
onAStick "hello"
```

If there are two generic parameters, the compiler will give them different names: `'a` for the first generic, `'b` for the second generic, and so on. Here's an example:

```
let concatString x y = x.ToString() + y.ToString()
```

The type signature for this has two generics: `'a` and `'b` :

```
val concatString : 'a -> 'b -> string
```

On the other hand, the compiler will recognize when only one generic type is required. In the following example, the x and y parameters must be of the same type:

```
let isEqual x y = (x=y)
```

So the function signature has the same generic type for both of them:

```
val isEqual : 'a -> 'a -> bool
```

Generic parameters are also very important when it comes to lists and more abstract structures, and we will be seeing them a lot in upcoming examples.

Other types

The types discussed so far are just the basic types. These types can be combined in various ways to make much more complex types. A full discussion of these types will have to wait for another series, but meanwhile, here is a brief introduction to them so that you can recognize them in function signatures.

- **The "tuple" types**. These are pairs, triples, etc., of other types. For example `("hello", 1)` is a tuple made from a string and an int. The comma is the distinguishing characteristic of a tuple -- if you see a comma in F#, it is almost certainly part of a tuple!

In function signatures, tuples are written as the "multiplication" of the two types involved. So in this case, the tuple would have type:

```
string * int      // ("hello", 1)
```

- **The collection types**. The most common of these are lists, sequences, and arrays. Lists and arrays are fixed size, while sequences are potentially infinite (behind the scenes, sequences are the same as `IEnumerable`). In function signatures, they have their own keywords: "`list`", "`seq`", and "`[]`" for arrays.

```
int list          // List type  e.g. [1;2;3]
string list       // List type  e.g. ["a";"b";"c"]
seq<int>          // Seq type   e.g. seq{1..10}
int []            // Array type e.g. [|1;2;3|]
```

- **The option type**. This is a simple wrapper for objects that might be missing. There are two cases: `Some` and `None`. In function signatures, they have their own "`option`" keyword:

```
int option        // Some(1)
```

- **The discriminated union type**. These are built from a set of choices of other types. We saw some examples of this in the "why use F#?" series. In function signatures, they are referred to by the name of the type, so there is no special keyword.
- **The record type**. These are like structures or database rows, a list of named slots. We saw some examples of this in the "why use F#?" series as well. In function signatures, they are referred to by the name of the type, so again there is no special keyword.

Test your understanding of types

How well do you understand the types yet? Here are some expressions for you -- see if you can guess their signatures. To see if you are correct, just run them in the interactive window!

```
let testA   = float 2
let testB x = float 2
let testC x = float 2 + x
let testD x = x.ToString().Length
let testE (x:float) = x.ToString().Length
let testF x = printfn "%s" x
let testG x = printfn "%f" x
let testH   = 2 * 2 |> ignore
let testI x = 2 * 2 |> ignore
let testJ (x:int) = 2 * 2 |> ignore
let testK   = "hello"
let testL() = "hello"
let testM x = x=x
let testN x = x 1          // hint: what kind of thing is x?
let testO x:string = x 1   // hint: what does :string modify?
```

Currying

After that little digression on basic types, we can turn back to functions again, and in particular the puzzle we mentioned earlier: if a mathematical function can only have one parameter, then how is it possible that an F# function can have more than one?

The answer is quite simple: a function with multiple parameters is rewritten as a series of new functions, each with only one parameter. And this is done automatically by the compiler for you. It is called "**currying**", after Haskell Curry, a mathematician who was an important influence on the development of functional programming.

To see how this works in practice, let's use a very basic example that prints two numbers:

```
//normal version
let printTwoParameters x y =
   printfn "x=%i y=%i" x y
```

Internally, the compiler rewrites it as something more like:

```
//explicitly curried version
let printTwoParameters x  =    // only one parameter!
   let subFunction y =
      printfn "x=%i y=%i" x y  // new function with one param
   subFunction                 // return the subfunction
```

Let's examine this in more detail:

1. Construct the function called " `printTwoParameters` " but with only *one* parameter: "x"
2. Inside that, construct a subfunction that has only *one* parameter: "y". Note that this inner function uses the "x" parameter but x is not passed to it explicitly as a parameter. The "x" parameter is in scope, so the inner function can see it and use it without needing it to be passed in.
3. Finally, return the newly created subfunction.
4. This returned function is then later used against "y". The "x" parameter is baked into it, so the returned function only needs the y param to finish off the function logic.

By rewriting it this way, the compiler has ensured that every function has only one parameter, as required. So when you use " `printTwoParameters` ", you might think that you are using a two parameter function, but it is actually only a one parameter function! You can see for yourself by passing only one argument instead of two:

```
// eval with one argument
printTwoParameters 1

// get back a function!
val it : (int -> unit) = <fun:printTwoParameters@286-3>
```

If you evaluate it with one argument, you don't get an error, you get back a function.

So what you are really doing when you call `printTwoParameters` with two arguments is:

- You call `printTwoParameters` with the first argument (x)
- `printTwoParameters` returns a new function that has "x" baked into it.
- You then call the new function with the second argument (y)

Here is an example of the step by step version, and then the normal version again.

```
// step by step version
let x = 6
let y = 99
let intermediateFn = printTwoParameters x  // return fn with
                                           // x "baked in"
let result  = intermediateFn y

// inline version of above
let result  = (printTwoParameters x) y

// normal version
let result  = printTwoParameters x y
```

Here is another example:

```
//normal version
let addTwoParameters x y =
   x + y

//explicitly curried version
let addTwoParameters x  =      // only one parameter!
   let subFunction y =
      x + y                    // new function with one param
   subFunction                 // return the subfunction

// now use it step by step
let x = 6
let y = 99
let intermediateFn = addTwoParameters x  // return fn with
                                         // x "baked in"
let result  = intermediateFn y

// normal version
let result  = addTwoParameters x y
```

Again, the "two parameter function" is actually a one parameter function that returns an intermediate function.

But wait a minute -- what about the " `+` " operation itself? It's a binary operation that must take two parameters, surely? No, it is curried like every other function. There is a function called " `+` " that takes one parameter and returns a new intermediate function, exactly like `addTwoParameters` above.

When we write the statement `x+y`, the compiler reorders the code to remove the infix and turns it into `(+) x y`, which is the function named `+` called with two parameters. Note that the function named "+" needs to have parentheses around it to indicate that it is being used as a normal function name rather than as an infix operator.

Finally, the two parameter function named `+` is treated as any other two parameter function would be.

```
// using plus as a single value function
let x = 6
let y = 99
let intermediateFn = (+) x     // return add with x baked in
let result  = intermediateFn y

// using plus as a function with two parameters
let result  = (+) x y

// normal version of plus as infix operator
let result  = x + y
```

And yes, this works for all other operators and built in functions like printf.

```
// normal version of multiply
let result  = 3 * 5

// multiply as a one parameter function
let intermediateFn = (*) 3   // return multiply with "3" baked in
let result  = intermediateFn 5

// normal version of printfn
let result  = printfn "x=%i y=%i" 3 5

// printfn as a one parameter function
let intermediateFn = printfn "x=%i y=%i" 3  // "3" is baked in
let result  = intermediateFn 5
```

Signatures of curried functions

Now that we know how curried functions work, what should we expect their signatures to look like?

Going back to the first example, " `printTwoParameters` ", we saw that it took one argument and returned an intermediate function. The intermediate function also took one argument and returned nothing (that is, unit). So the intermediate function has type `int->unit` . In other words, the domain of `printTwoParameters` is `int` and the range is `int->unit` . Putting this together we see that the final signature is:

```
val printTwoParameters : int -> (int -> unit)
```

If you evaluate the explicitly curried implementation, you will see the parentheses in the signature, as written above, but if you evaluate the normal implementation, which is implicitly curried, the parentheses are left off, like so:

```
val printTwoParameters : int -> int -> unit
```

The parentheses are optional. If you are trying to make sense of function signatures it might be helpful to add them back in mentally.

At this point you might be wondering, what is the difference between a function that returns an intermediate function and a regular two parameter function?

Here's a one parameter function that returns a function:

```
let add1Param x = (+) x
// signature is = int -> (int -> int)
```

Here's a two parameter function that returns a simple value:

```
let add2Params x y = (+) x y
// signature is = int -> int -> int
```

The signatures are slightly different, but in practical terms, there *is* no difference*, only that the second function is automatically curried for you.

Functions with more than two parameters

How does currying work for functions with more than two parameters? Exactly the same way: for each parameter except the last one, the function returns an intermediate function with the previous parameters baked in.

Consider this contrived example. I have explicitly specified the types of the parameters, but the function itself does nothing.

```
let multiParamFn (p1:int)(p2:bool)(p3:string)(p4:float)=
   ()   //do nothing

let intermediateFn1 = multiParamFn 42
   // intermediateFn1 takes a bool
   // and returns a new function (string -> float -> unit)
let intermediateFn2 = intermediateFn1 false
   // intermediateFn2 takes a string
   // and returns a new function (float -> unit)
let intermediateFn3 = intermediateFn2 "hello"
   // intermediateFn3 takes a float
   // and returns a simple value (unit)
let finalResult = intermediateFn3 3.141
```

The signature of the overall function is:

```
val multiParamFn : int -> bool -> string -> float -> unit
```

and the signatures of the intermediate functions are:

```
val intermediateFn1 : (bool -> string -> float -> unit)
val intermediateFn2 : (string -> float -> unit)
val intermediateFn3 : (float -> unit)
val finalResult : unit = ()
```

A function signature can tell you how many parameters the function takes: just count the number of arrows outside of parentheses. If the function takes or returns other function parameters, there will be other arrows in parentheses, but these can be ignored. Here are some examples:

```
int->int->int      // two int parameters and returns an int

string->bool->int  // first param is a string, second is a bool,
                   // returns an int

int->string->bool->unit // three params (int,string,bool)
                        // returns nothing (unit)

(int->string)->int      // has only one parameter, a function
                        // value (from int to string)
                        // and returns a int

(int->string)->(int->bool) // takes a function (int to string)
                           // returns a function (int to bool)
```

Issues with multiple parameters

The logic behind currying can produce some unexpected results until you understand it. Remember that you will not get an error if you evaluate a function with fewer arguments than it is expecting. Instead you will get back a partially applied function. If you then go on to use this partially applied function in a context where you expect a value, you will get obscure error messages from the compiler.

Here's an innocuous looking function:

```
// create a function
let printHello() = printfn "hello"
```

What would you expect to happen when we call it as shown below? Will it print "hello" to the console? Try to guess before evaluating it, and here's a hint: be sure to take a look at the function signature.

```
// call it
printHello
```

It will *not* be called as expected. The original function expects a unit argument that was not supplied, so you are getting a partially applied function (in this case with no arguments).

How about this? Will it compile?

```
let addXY x y =
    printfn "x=%i y=%i" x
    x + y
```

If you evaluate it, you will see that the compiler complains about the printfn line.

```
printfn "x=%i y=%i" x
//^^^^^^^^^^^^^^^^^^^^^
//warning FS0193: This expression is a function value, i.e. is missing
//arguments. Its type is  ^a -> unit.
```

If you didn't understand currying, this message would be very cryptic! All expressions that are evaluated standalone like this (i.e. not used as a return value or bound to something with "let") *must* evaluate to the unit value. And in this case, it is does *not* evaluate to the unit value, but instead evaluates to a function. This is a long winded way of saying that `printfn` is missing an argument.

A common case of errors like this is when interfacing with the .NET library. For example, the `ReadLine` method of a `TextReader` must take a unit parameter. It is often easy to forget this and leave off the parens, in which case you do not get a compiler error immediately, but only when you try to treat the result as a string.

```
let reader = new System.IO.StringReader("hello");

let line1 = reader.ReadLine        // wrong but compiler doesn't
                                   // complain
printfn "The line is %s" line1     //compiler error here!
// ==> error FS0001: This expression was expected to have
// type string but here has type unit -> string

let line2 = reader.ReadLine()      //correct
printfn "The line is %s" line2     //no compiler error
```

In the code above, `line1` is just a pointer or delegate to the `Readline` method, not the string that we expected. The use of `()` in `reader.ReadLine()` actually executes the function.

Too many parameters

You can get similar cryptic messages when you have too many parameters as well. Here are some examples of passing too many parameters to printf.

```
printfn "hello" 42
// ==> error FS0001: This expression was expected to have
//                   type 'a -> 'b but here has type unit

printfn "hello %i" 42 43
// ==> Error FS0001: Type mismatch. Expecting a 'a -> 'b -> 'c
//                   but given a 'a -> unit

printfn "hello %i %i" 42 43 44
// ==> Error FS0001: Type mismatch. Expecting a 'a->'b->'c->'d
//                   but given a 'a -> 'b -> unit
```

For example, in the last case, the compiler is saying that it expects the format argument to have three parameters (the signature `'a -> 'b -> 'c -> 'd` has three parameters) but it is given only two (the signature `'a -> 'b -> unit` has two parameters).

In cases not using `printf`, passing too many parameters will often mean that you end up with a simple value that you then try to pass a parameter to. The compiler will complain that the simple value is not a function.

```
let add1 x = x + 1
let x = add1 2 3
// ==>   error FS0003: This value is not a function
//                     and cannot be applied
```

If you break the call into a series of explicit intermediate functions, as we did earlier, you can see exactly what is going wrong.

```
let add1 x = x + 1
let intermediateFn = add1 2   //returns a simple value
let x = intermediateFn 3      //intermediateFn is not a function!
// ==>   error FS0003: This value is not a function
//                     and cannot be applied
```

Partial application

In the previous post on currying, we looked at breaking multiple parameter functions into smaller one parameter functions. It is the mathematically correct way of doing it, but that is not the only reason it is done -- it also leads to a very powerful technique called **partial function application**. This is a very widely used style in functional programming, and it is important to understand it.

The idea of partial application is that if you fix the first N parameters of the function, you get a function of the remaining parameters. From the discussion on currying, you can probably see how this comes about naturally.

Here are some simple examples that demonstrate this:

```
// create an "adder" by partial application of add
let add42 = (+) 42    // partial application
add42 1
add42 3

// create a new list by applying the add42 function
// to each element
[1;2;3] |> List.map add42

// create a "tester" by partial application of "less than"
let twoIsLessThan = (<) 2   // partial application
twoIsLessThan 1
twoIsLessThan 3

// filter each element with the twoIsLessThan function
[1;2;3] |> List.filter twoIsLessThan

// create a "printer" by partial application of printfn
let printer = printfn "printing param=%i"

// loop over each element and call the printer function
[1;2;3] |> List.iter printer
```

In each case, we create a partially applied function that we can then reuse in multiple contexts.

The partial application can just as easily involve fixing function parameters, of course. Here are some examples:

```
// an example using List.map
let add1 = (+) 1
let add1ToEach = List.map add1   // fix the "add1" function

// test
add1ToEach [1;2;3;4]

// an example using List.filter
let filterEvens =
   List.filter (fun i -> i%2 = 0) // fix the filter function

// test
filterEvens [1;2;3;4]
```

The following more complex example shows how the same approach can be used to create "plug in" behavior that is transparent.

- We create a function that adds two numbers, but in addition takes a logging function that will log the two numbers and the result.
- The logging function has two parameters: (string) "name" and (generic) "value", so it has signature `string->'a->unit` .
- We then create various implementations of the logging function, such as a console logger or a popup logger.
- And finally we partially apply the main function to create new functions that have a particular logger baked into them.

```
// create an adder that supports a pluggable logging function
let adderWithPluggableLogger logger x y =
    logger "x" x
    logger "y" y
    let result = x + y
    logger "x+y"  result
    result

// create a logging function that writes to the console
let consoleLogger argName argValue =
    printfn "%s=%A" argName argValue

//create an adder with the console logger partially applied
let addWithConsoleLogger = adderWithPluggableLogger consoleLogger
addWithConsoleLogger  1 2
addWithConsoleLogger  42 99

// create a logging function that creates popup windows
let popupLogger argName argValue =
    let message = sprintf "%s=%A" argName argValue
    System.Windows.Forms.MessageBox.Show(
                                 text=message,caption="Logger")
      |> ignore

//create an adder with the popup logger partially applied
let addWithPopupLogger  = adderWithPluggableLogger popupLogger
addWithPopupLogger  1 2
addWithPopupLogger  42 99
```

These functions with the logger baked in can in turn be used like any other function. For example, we can create a partial application to add 42, and then pass that into a list function, just like we did for the simple " `add42` " function.

```
// create a another adder with 42 baked in
let add42WithConsoleLogger = addWithConsoleLogger 42
[1;2;3] |> List.map add42WithConsoleLogger
[1;2;3] |> List.map add42               //compare without logger
```

These partially applied functions are a very useful tool. We can create library functions which are flexible (but complicated), yet make it easy to create reusable defaults so that callers don't have to be exposed to the complexity all the time.

Designing functions for partial application

You can see that the order of the parameters can make a big difference in the ease of use for partial application. For example, most of the functions in the `List` library such as `List.map` and `List.filter` have a similar form, namely:

```
List-function [function parameter(s)] [list]
```

The list is always the last parameter. Here are some examples of the full form:

```
List.map    (fun i -> i+1) [0;1;2;3]
List.filter (fun i -> i>1) [0;1;2;3]
List.sortBy (fun i -> -i ) [0;1;2;3]
```

And the same examples using partial application:

```
let eachAdd1 = List.map (fun i -> i+1)
eachAdd1 [0;1;2;3]

let excludeOneOrLess = List.filter (fun i -> i>1)
excludeOneOrLess [0;1;2;3]

let sortDesc = List.sortBy (fun i -> -i)
sortDesc [0;1;2;3]
```

If the library functions were written with the parameters in a different order, it would be much more inconvenient to use them with partial application.

As you write your own multi-parameter functions, you might wonder what the best parameter order is. As with all design questions, there is no "right" answer to this question, but here are some commonly accepted guidelines:

1. Put earlier: parameters more likely to be static
2. Put last: the data structure or collection (or most varying argument)
3. For well-known operations such as "subtract", put in the expected order

Guideline 1 is straightforward. The parameters that are most likely to be "fixed" with partial application should be first. We saw this with the logger example earlier.

Guideline 2 makes it easier to pipe a structure or collection from function to function. We have seen this many times already with list functions.

```
// piping using list functions
let result =
   [1..10]
   |> List.map (fun i -> i+1)
   |> List.filter (fun i -> i>5)
```

Similarly, partially applied list functions are easy to compose, because the list parameter itself can be easily elided:

```
let compositeOp = List.map (fun i -> i+1)
                  >> List.filter (fun i -> i>5)
let result = compositeOp [1..10]
```

Wrapping BCL functions for partial application

The .NET base class library functions are easy to access in F#, but are not really designed for use with a functional language like F#. For example, most functions have the data parameter first, while with F#, as we have seen, the data parameter should normally come last.

However, it is easy enough to create wrappers for them that are more idiomatic. For example, in the snippet below, the .NET string functions are rewritten to have the string target be the last parameter rather than the first:

```
// create wrappers for .NET string functions
let replace oldStr newStr (s:string) =
  s.Replace(oldValue=oldStr, newValue=newStr)

let startsWith lookFor (s:string) =
  s.StartsWith(lookFor)
```

Once the string becomes the last parameter, we can then use them with pipes in the expected way:

```
let result =
     "hello"
     |> replace "h" "j"
     |> startsWith "j"

["the"; "quick"; "brown"; "fox"]
     |> List.filter (startsWith "f")
```

or with function composition:

```
let compositeOp = replace "h" "j" >> startsWith "j"
let result = compositeOp "hello"
```

Understanding the "pipe" function

Now that you have seen how partial application works, you should be able to understand how the "pipe" function works.

The pipe function is defined as:

```
let (|>) x f = f x
```

All it does is allow you to put the function argument in front of the function rather than after. That's all.

```
let doSomething x y z = x+y+z
doSomething 1 2 3       // all parameters after function
```

If the function has multiple parameters, then it appears that the input is the final parameter. Actually what is happening is that the function is partially applied, returning a function that has a single parameter: the input

Here's the same example rewritten to use partial application

```
let doSomething x y  =
   let intermediateFn z = x+y+z
   intermediateFn        // return intermediateFn

let doSomethingPartial = doSomething 1 2
doSomethingPartial 3     // only one parameter after function now
3 |> doSomethingPartial  // same as above - last parameter piped in
```

As you have already seen, the pipe operator is extremely common in F#, and used all the time to preserve a natural flow. Here are some more usages that you might see:

```
"12" |> int               // parses string "12" to an int
1 |> (+) 2 |> (*) 3       // chain of arithmetic
```

The reverse pipe function

You might occasionally see the reverse pipe function "<|" being used.

```
let (<|) f x = f x
```

It seems that this function doesn't really do anything different from normal, so why does it exist?

The reason is that, when used in the infix style as a binary operator, it reduces the need for parentheses and can make the code cleaner.

```
printf "%i" 1+2          // error
printf "%i" (1+2)        // using parens
printf "%i" <| 1+2       // using reverse pipe
```

You can also use piping in both directions at once to get a pseudo infix notation.

```
let add x y = x + y
(1+2) add (3+4)          // error
1+2 |> add <| 3+4        // pseudo infix
```

Function associativity and composition

Function associativity

If we have a chain of functions in a row, how are they combined?

For example, what does this mean?

```
let F x y z = x y z
```

Does it mean apply the function y to the argument z, and then take the result and use it as an argument for x? In which case it is the same as:

```
let F x y z = x (y z)
```

Or does it mean apply the function x to the argument y, and then take the resulting function and evaluate it with the argument z? In which case it is the same as:

```
let F x y z = (x y) z
```

The answer is the latter. Function application is *left associative*. That is, evaluating `x y z` is the same as evaluating `(x y) z`. And evaluating `w x y z` is the same as evaluating `((w x) y) z`. This should not be a surprise. We have already seen that this is how partial application works. If you think of x as a two parameter function, then `(x y) z` is the result of partial application of the first parameter, followed by passing the z argument to the intermediate function.

If you do want to do right association, you can use explicit parentheses, or you can use a pipe. The following three forms are equivalent.

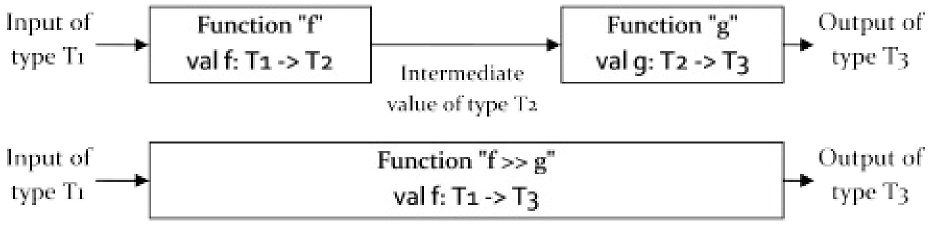
```
let F x y z = x (y z)
let F x y z = y z |> x    // using forward pipe
let F x y z = x <| y z    // using backward pipe
```

As an exercise, work out the signatures for these functions without actually evaluating them!

Function composition

We've mentioned function composition a number of times in passing now, but what does it actually mean? It can seem quite intimidating at first, but it is actually quite simple.

Say that you have a function "f" that maps from type "T1" to type "T2", and say that you also have a function "g" that maps from type "T2" to type "T3". Then you can connect the output of "f" to the input of "g", creating a new function that maps from type "T1" to type "T3".



Here's an example

```
let f (x:int) = float x * 3.0  // f is int->float
let g (x:float) = x > 4.0      // g is float->bool
```

We can create a new function h that takes the output of "f" and uses it as the input for "g".

```
let h (x:int) =
    let y = f(x)
    g(y)                   // return output of g
```

A much more compact way is this:

```
let h (x:int) = g ( f(x) ) // h is int->bool

//test
h 1
h 2
```

So far, so straightforward. What is interesting is that we can define a new function called "compose" that, given functions "f" and "g", combines them in this way without even knowing their signatures.

```
let compose f g x = g ( f(x) )
```

If you evaluate this, you will see that the compiler has correctly deduced that if " `f` " is a function from generic type `'a` to generic type `'b` , then " `g` " is constrained to have generic type `'b` as an input. And the overall signature is:

```
val compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
```

(Note that this generic composition operation is only possible because every function has one input and one output. This approach would not be possible in a non-functional language.)

As we have seen, the actual definition of compose uses the " `>>` " symbol.

```
let (>>) f g x = g ( f(x) )
```

Given this definition, we can now use composition to build new functions from existing ones.

```
let add1 x = x + 1
let times2 x = x * 2
let add1Times2 x = (>>) add1 times2 x

//test
add1Times2 3
```

This explicit style is quite cluttered. We can do a few things to make it easier to use and understand.

First, we can leave off the x parameter so that the composition operator returns a partial application.

```
let add1Times2 = (>>) add1 times2
```

And now we have a binary operation, so we can put the operator in the middle.

```
let add1Times2 = add1 >> times2
```

And there you have it. Using the composition operator allows code to be cleaner and more straightforward.

```
let add1 x = x + 1
let times2 x = x * 2

//old style
let add1Times2 x = times2(add1 x)

//new style
let add1Times2 = add1 >> times2
```

Using the composition operator in practice

The composition operator (like all infix operators) has lower precedence than normal function application. This means that the functions used in composition can have arguments without needing to use parentheses.

For example, if the "add" and "times" functions have an extra parameter, this can be passed in during the composition.

```
let add n x = x + n
let times n x = x * n
let add1Times2 = add 1 >> times 2
let add5Times3 = add 5 >> times 3

//test
add5Times3 1
```

As long as the inputs and outputs match, the functions involved can use any kind of value. For example, consider the following, which performs a function twice:

```
let twice f = f >> f    //signature is ('a -> 'a) -> ('a -> 'a)
```

Note that the compiler has deduced that the function f must use the same type for both input and output.

Now consider a function like " `+` ". As we have seen earlier, the input is an `int`, but the output is actually a partially applied function `(int->int)`. The output of " `+` " can thus be used as the input of " `twice` ". So we can write something like:

```
let add1 = (+) 1           // signature is (int -> int)
let add1Twice = twice add1 // signature is also (int -> int)

//test
add1Twice 9
```

On the other hand, we can't write something like:

```
let addThenMultiply = (+) >> (*)
```

because the input to "*" must be an `int` value, not an `int->int` function (which is what the output of addition is).

But if we tweak it so that the first function has an output of just `int` instead, then it does work:

```
let add1ThenMultiply = (+) 1 >> (*)
// (+) 1 has signature (int -> int) and output is an 'int'

//test
add1ThenMultiply 2 7
```

Composition can also be done backwards using the " `<<` " operator, if needed.

```
let times2Add1 = add 1 << times 2
times2Add1 3
```

Reverse composition is mainly used to make code more English-like. For example, here is a simple example:

```
let myList = []
myList |> List.isEmpty |> not    // straight pipeline

myList |> (not << List.isEmpty)  // using reverse composition
```

Composition vs. pipeline

At this point, you might be wondering what the difference is between the composition operator and the pipeline operator, as they can seem quite similar.

First let's look again at the definition of the pipeline operator:

```
let (|>) x f = f x
```

All it does is allow you to put the function argument in front of the function rather than after. That's all. If the function has multiple parameters, then the input would be the final parameter. Here's the example used earlier.

```
let doSomething x y z = x+y+z
doSomething 1 2 3       // all parameters after function
3 |> doSomething 1 2    // last parameter piped in
```

Composition is not the same thing and cannot be a substitute for a pipe. In the following case the number 3 is not even a function, so its "output" cannot be fed into `doSomething`:

```
3 >> doSomething 1 2     // not allowed
// f >> g is the same as  g(f(x)) so rewriting it we have:
doSomething 1 2 ( 3(x) ) // implies 3 should be a function!
// error FS0001: This expression was expected to have type 'a->'b
//                but here has type int
```

The compiler is complaining that "3" should be some sort of function `'a->'b`.

Compare this with the definition of composition, which takes 3 arguments, where the first two must be functions.

```
let (>>) f g x = g ( f(x) )

let add n x = x + n
let times n x = x * n
let add1Times2 = add 1 >> times 2
```

Trying to use a pipe instead doesn't work. In the following example, " `add 1` " is a (partial) function of type `int->int`, and cannot be used as the second parameter of " `times 2` ".

```
let add1Times2 = add 1 |> times 2   // not allowed
// x |> f is the same as  f(x) so rewriting it we have:
let add1Times2 = times 2 (add 1)    // add1 should be an int
// error FS0001: Type mismatch. 'int -> int' does not match 'int'
```

The compiler is complaining that " `times 2` " should take an `int->int` parameter, that is, be of type `(int->int)->'a`.

Defining functions

We have seen how to create typical functions using the "let" syntax, below:

```
let add x y = x + y
```

In this section, we'll look at some other ways of creating functions, and tips for defining functions.

Anonymous functions (a.k.a. lambdas)

If you are familiar with lambdas in other languages, this will not be new to you. An anonymous function (or "lambda expression") is defined using the form:

```
fun parameter1 parameter2 etc -> expression
```

If you are used to lambdas in C# there are a couple of differences:

- the lambda must have the special keyword `fun`, which is not needed in the C# version
- the arrow symbol is a single arrow `->` rather than the double arrow (`=>`) in C#.

Here is a lambda that defines addition:

```
let add = fun x y -> x + y
```

This is exactly the same as a more conventional function definition:

```
let add x y = x + y
```

Lambdas are often used when you have a short expression and you don't want to define a function just for that expression. This is particularly common with list operations, as we have seen already.

```
// with separately defined function
let add1 i = i + 1
[1..10] |> List.map add1

// inlined without separately defined function
[1..10] |> List.map (fun i -> i + 1)
```

Note that you must use parentheses around the lambda.

Lambdas are also used when you want to make it clear that you are returning a function from another function. For example, the "`adderGenerator`" function that we talked about earlier could be rewritten with a lambda.

```
// original definition
let adderGenerator x = (+) x

// definition using lambda
let adderGenerator x = fun y -> x + y
```

The lambda version is slightly longer, but makes it clear that an intermediate function is being returned.

You can nest lambdas as well. Here is yet another definition of `adderGenerator`, this time using lambdas only.

```
let adderGenerator = fun x -> (fun y -> x + y)
```

Can you see that all three of the following definitions are the same thing?

```
let adderGenerator1 x y = x + y
let adderGenerator2 x   = fun y -> x + y
let adderGenerator3     = fun x -> (fun y -> x + y)
```

If you can't see it, then do reread the post on currying. This is important stuff to understand!

Pattern matching on parameters

When defining a function, you can pass an explicit parameter, as we have seen, but you can also pattern match directly in the parameter section. In other words, the parameter section can contain *patterns*, not just identifiers!

The following example demonstrates how to use patterns in a function definition:

```
type Name = {first:string; last:string} // define a new type
let bob = {first="bob"; last="smith"}   // define a value

// single parameter style
let f1 name =                       // pass in single parameter
   let {first=f; last=l} = name     // extract in body of function
   printfn "first=%s; last=%s" f l

// match in the parameter itself
let f2 {first=f; last=l} =          // direct pattern matching
   printfn "first=%s; last=%s" f l

// test
f1 bob
f2 bob
```

This kind of matching can only occur when the matching is always possible. For example, you cannot match on union types or lists this way, because some cases might not be matched.

```
let f3 (x::xs) =            // use pattern matching on a list
   printfn "first element is=%A" x
```

You will get a warning about incomplete pattern matches.

A common mistake: tuples vs. multiple parameters

If you come from a C-like language, a tuple used as a single function parameter can look awfully like multiple parameters. They are not the same thing at all! As I noted earlier, if you see a comma, it is probably part of a tuple. Parameters are separated by spaces.

Here is an example of the confusion:

```
// a function that takes two distinct parameters
let addTwoParams x y = x + y

// a function that takes a single tuple parameter
let addTuple aTuple =
   let (x,y) = aTuple
   x + y

// another function that takes a single tuple parameter
// but looks like it takes two ints
let addConfusingTuple (x,y) = x + y
```

- The first definition, " `addTwoParams` ", takes two parameters, separated with spaces.
- The second definition, " `addTuple` ", takes a single parameter. It then binds "x" and "y" to the inside of the tuple and does the addition.
- The third definition, " `addConfusingTuple` ", takes a single parameter just like " `addTuple` ", but the tricky thing is that the tuple is unpacked and bound as part of the parameter definition using pattern matching. Behind the scenes, it is exactly the same as " `addTuple` ".

Let's look at the signatures (it is always a good idea to look at the signatures if you are unsure)

```
val addTwoParams : int -> int -> int        // two params
val addTuple : int * int -> int             // tuple->int
val addConfusingTuple : int * int -> int    // tuple->int
```

Now let's use them:

```
//test
addTwoParams 1 2      // ok - uses spaces to separate args
addTwoParams (1,2)    // error trying to pass a single tuple
//   => error FS0001: This expression was expected to have type
//                    int but here has type 'a * 'b
```

Here we can see an error occur in the second case above.

First, the compiler treats `(1,2)` as a generic tuple of type `('a * 'b)`, which it attempts to pass as the first parameter to " `addTwoParams` ". Then it complains that the first parameter of `addTwoParams` is an `int`, and we're trying to pass a tuple.

To make a tuple, use a comma! Here's how to do it correctly:

```
addTuple (1,2)           // ok
addConfusingTuple (1,2)  // ok

let x = (1,2)
addTuple x               // ok

let y = 1,2              // it's the comma you need,
                         // not the parentheses!
addTuple y               // ok
addConfusingTuple y      // ok
```

Conversely, if you attempt to pass multiple arguments to a function expecting a tuple, you will also get an obscure error.

```
addConfusingTuple 1 2    // error trying to pass two args
// => error FS0003: This value is not a function and
//                  cannot be applied
```

In this case, the compiler thinks that, since you are passing two arguments, `addConfusingTuple` must be curryable. So then " `addConfusingTuple 1` " would be a partial application that returns another intermediate function. Trying to apply that intermediate function with "2" gives an error, because there is no intermediate function! We saw this exact same error in the post on currying, when we discussed the issues that can occur from having too many parameters.

Why not use tuples as parameters?

The discussion of the issues with tuples above shows that there's another way to define functions with more than one parameter: rather than passing them in separately, all the parameters can be combined into a single composite data structure. In the example below, the function takes a single parameter, which is a tuple containing three items.

```
let f (x,y,z) = x + y * z
// type is int * int * int -> int

// test
f (1,2,3)
```

Note that the function signature is different from a true three parameter function. There is only one arrow, so only one parameter, and the stars indicate that this is a tuple of `(int*int*int)` .

When would we want to use tuple parameters instead of individual ones?

- When the tuples are meaningful in themselves. For example, if we are working with three dimensional coordinates, a three-tuple might well be more convenient than three separate dimensions.
- Tuples are occasionally used to bundle data together in a single structure that should be kept together. For example, the `TryParse` functions in .NET library return the result and a Boolean as a tuple. But if you have a lot of data that is kept together as a bundle, then you will probably want to define a record or class type to store it.

A special case: tuples and .NET library functions

One area where commas are seen a lot is when calling .NET library functions!

These all take tuple-like arguments, and so these calls look just the same as they would from C#:

```
// correct
System.String.Compare("a","b")

// incorrect
System.String.Compare "a" "b"
```

The reason is that .NET library functions are not curried and cannot be partially applied. *All* the parameters must *always* be passed in, and using a tuple-like approach is the obvious way to do this.

But do note that although these calls look like tuples, they are actually a special case. Real tuples cannot be used, so the following code is invalid:

```
let tuple = ("a","b")
System.String.Compare tuple   // error

System.String.Compare "a","b" // error
```

If you do want to partially apply .NET library functions, it is normally trivial to write wrapper functions for them, as we have seen earlier, and as shown below:

```
// create a wrapper function
let strCompare x y = System.String.Compare(x,y)

// partially apply it
let strCompareWithB = strCompare "B"

// use it with a higher order function
["A";"B";"C"]
|> List.map strCompareWithB
```

Guidelines for separate vs. grouped parameters

The discussion on tuples leads us to a more general topic: when should function parameters be separate and when should they be grouped?

Note that F# is different from C# in this respect. In C# *all* the parameters are *always* provided, so the question does not even arise! In F#, due to partial application, only some parameters might be provided, so you need to distinguish between those that are required to be grouped together vs. those that are independent.

Here are some general guidelines of how to structure parameters when you are designing your own functions.

- In general, it is always better to use separate parameters rather than passing them as a single structure such as a tuple or record. This allows for more flexible behavior such as partial application.
- But, when a group of parameters *must* all be set at once, then *do* use some sort of grouping mechanism.

In other words, when designing a function, ask yourself "could I provide this parameter in isolation?" If the answer is no, the parameters should be grouped.

Let's look at some examples:

```
// Pass in two numbers for addition.
// The numbers are independent, so use two parameters
let add x y = x + y

// Pass in two numbers as a geographical co-ordinate.
// The numbers are dependent, so group them into a tuple or record
let locateOnMap (xCoord,yCoord) = // do something

// Set first and last name for a customer.
// The values are dependent, so group them into a record.
type CustomerName = {First:string; Last:string}
let setCustomerName aCustomerName = // good
let setCustomerName first last = // not recommended

// Set first and last name and and pass the
// authorizing credentials as well.
// The name and credentials are independent, keep them separate
let setCustomerName myCredentials aName = //good
```

Finally, do be sure to order the parameters appropriately to assist with partial application (see the guidelines in the earlier post). For example, in the last function above, why did I put the `myCredentials` parameter ahead of the `aName` parameter?

Parameter-less functions

Sometimes we may want functions that don't take any parameters at all. For example, we may want a "hello world" function that we can call repeatedly. As we saw in a previous section, the naive definition will not work.

```
let sayHello = printfn "Hello World!"     // not what we want
```

The fix is to add a unit parameter to the function, or use a lambda.

```
let sayHello() = printfn "Hello World!"           // good
let sayHello = fun () -> printfn "Hello World!"   // good
```

And then the function must always be called with a unit argument:

```
// call it
sayHello()
```

This is particularly common with the .NET libraries. Some examples are:

```
Console.ReadLine()
System.Environment.GetCommandLineArgs()
System.IO.Directory.GetCurrentDirectory()
```

Do remember to call them with the unit parameter!

Defining new operators

You can define functions named using one or more of the operator symbols (see the F# documentation for the exact list of symbols that you can use):

```
// define
let (.*%) x y = x + y + 1
```

You must use parentheses around the symbols when defining them.

Note that for custom operators that begin with `*`, a space is required; otherwise the `(*` is interpreted as the start of a comment:

```
let ( *+* ) x y = x + y + 1
```

Once defined, the new function can be used in the normal way, again with parens around the symbols:

```
let result = (.*%) 2 3
```

If the function has exactly two parameters, you can use it as an infix operator without parentheses.

```
let result = 2 .*% 3
```

You can also define prefix operators that start with `!` or `~` (with some restrictions -- see the F# documentation on operator overloading)

```
let (~%%) (s:string) = s.ToCharArray()

//use
let result = %% "hello"
```

In F# it is quite common to create your own operators, and many libraries will export operators with names such as `>=>` and `<*>`.

Point-free style

We have already seen many examples of leaving off the last parameter of functions to reduce clutter. This style is referred to as **point-free style** or **tacit programming**.

Here are some examples:

```
let add x y = x + y   // explicit
let add x = (+) x     // point free

let add1Times2 x = (x + 1) * 2    // explicit
let add1Times2 = (+) 1 >> (*) 2   // point free

let sum list = List.reduce (fun sum e -> sum+e) list // explicit
let sum = List.reduce (+)                            // point free
```

There are pros and cons to this style.

On the plus side, it focuses attention on the high level function composition rather than the low level objects. For example "`(+) 1 >> (*) 2`" is clearly an addition operation followed by a multiplication. And "`List.reduce (+)`" makes it clear that the plus operation is key, without needing to know about the list it is actually applied to.

Point-free helps to clarify the underlying algorithm and reveal commonalities between code -- the "`reduce`" function used above is a good example of this -- it will be discussed in a planned series on list processing.

On the other hand, too much point-free style can make for confusing code. Explicit parameters can act as a form of documentation, and their names (such as "list") make it clear what the function is acting on.

As with anything in programming, the best guideline is to use the approach that provides the most clarity.

Combinators

The word "**combinator**" is used to describe functions whose result depends only on their parameters. That means there is no dependency on the outside world, and in particular no other functions or global value can be accessed at all.

In practice, this means that a combinator function is limited to combining its parameters in various ways.

We have already seen some combinators already: the "pipe" operator and the "compose" operator. If you look at their definitions, it is clear that all they do is reorder the parameters in various ways

```
let (|>) x f = f x             // forward pipe
let (<|) f x = f x             // reverse pipe
let (>>) f g x = g (f x)       // forward composition
let (<<) g f x = g (f x)       // reverse composition
```

On the other hand, a function like "printf", although primitive, is not a combinator, because it has a dependency on the outside world (I/O).

Combinator birds

Combinators are the basis of a whole branch of logic (naturally called "combinatory logic") that was invented many years before computers and programming languages. Combinatory logic has had a very large influence on functional programming.

To read more about combinators and combinatory logic, I recommend the book "To Mock a Mockingbird" by Raymond Smullyan. In it, he describes many other combinators and whimsically gives them names of birds. Here are some examples of some standard combinators and their bird names:

```
let I x = x                // identity function, or the Idiot bird
let K x y = x              // the Kestrel
let M x = x >> x           // the Mockingbird
let T x y = y x            // the Thrush (this looks familiar!)
let Q x y z = y (x z)      // the Queer bird (also familiar!)
let S x y z = x z (y z)    // The Starling
// and the infamous...
let rec Y f x = f (Y f) x  // Y-combinator, or Sage bird
```

The letter names are quite standard, so if you refer to "the K combinator", everyone will be familiar with that terminology.

It turns out that many common programming patterns can be represented using these standard combinators. For example, the Kestrel is a common pattern in fluent interfaces where you do something but then return the original object. The Thrush is the pipe operation, the Queer bird is forward composition, and the Y-combinator is famously used to make functions recursive.

Indeed, there is a well-known theorem that states that any computable function whatsoever can be built from just two basic combinators, the Kestrel and the Starling.

Combinator libraries

A combinator library is a code library that exports a set of combinator functions that are designed to work together. The user of the library can then easily combine simple functions together to make bigger and more complex functions, like building with Lego.

A well designed combinator library allows you to focus on the high level operations, and push the low level "noise" to the background. We've already seen some examples of this power in the examples in "why use F#" series, and the `List` module is full of them -- the "`fold`" and "`map`" functions are also combinators, if you think about it.

Another advantage of combinators is that they are the safest type of function. As they have no dependency on the outside world they cannot change if the global environment changes. A function that reads a global value or uses a library function can break or alter between calls if the context is different. This can never happen with combinators.

In F#, combinator libraries are available for parsing (the FParsec library), HTML construction, testing frameworks, and more. We'll discuss and use combinators further in later series.

Recursive functions

Often, a function will need to refer to itself in its body. The classic example is the Fibonacci function:

```
let fib i =
   match i with
   | 1 -> 1
   | 2 -> 1
   | n -> fib(n-1) + fib(n-2)
```

Unfortunately, this will not compile:

```
error FS0039: The value or constructor 'fib' is not defined
```

You have to tell the compiler that this is a recursive function using the rec keyword.

```
let rec fib i =
   match i with
   | 1 -> 1
   | 2 -> 1
   | n -> fib(n-1) + fib(n-2)
```

Recursive functions and data structures are extremely common in functional programming, and I hope to devote a whole later series to this topic.

Function signatures

It may not be obvious, but F# actually has two syntaxes - one for normal (value) expressions, and one for type definitions. For example:

```
[1;2;3]      // a normal expression
int list     // a type expression

Some 1       // a normal expression
int option   // a type expression

(1,"a")      // a normal expression
int * string // a type expression
```

Type expressions have a special syntax that is *different* from the syntax used in normal expressions. You have already seen many examples of this when you use the interactive session, because the type of each expression has been printed along with its evaluation.

As you know, F# uses type inference to deduce types, so you don't often need to explicitly specify types in your code, especially for functions. But in order to work effectively in F#, you *do* need to understand the type syntax, so that you can build your own types, debug type errors, and understand function signatures. In this post, we'll focus on its use in function signatures.

Here are some example function signatures using the type syntax:

```
// expression syntax          // type syntax
let add1 x = x + 1            // int -> int
let add x y = x + y           // int -> int -> int
let print x = printf "%A" x   // 'a -> unit
System.Console.ReadLine       // unit -> string
List.sum                      // 'a list -> 'a
List.filter                   // ('a -> bool) -> 'a list -> 'a list
List.map                      // ('a -> 'b) -> 'a list -> 'b list
```

Understanding functions through their signatures

Just by examining a function's signature, you can often get some idea of what it does. Let's look at some examples and analyze them in turn.

```
// function signature 1
int -> int -> int
```

This function takes two `int` parameters and returns another, so presumably it is some sort of mathematical function such as addition, subtraction, multiplication, or exponentiation.

```
// function signature 2
int -> unit
```

This function takes an `int` and returns a `unit`, which means that the function is doing something important as a side-effect. Since there is no useful return value, the side effect is probably something to do with writing to IO, such as logging, writing to a file or database, or something similar.

```
// function signature 3
unit -> string
```

This function takes no input but returns a `string`, which means that the function is conjuring up a string out of thin air! Since there is no explicit input, the function probably has something to do with reading (from a file say) or generating (a random string, say).

```
// function signature 4
int -> (unit -> string)
```

This function takes an `int` input and returns a function that when called, returns strings. Again, the function probably has something to do with reading or generating. The input probably initializes the returned function somehow. For example, the input could be a file handle, and the returned function something like `readline()`. Or the input could be a seed for a random string generator. We can't tell exactly, but we can make some educated guesses.

```
// function signature 5
'a list -> 'a
```

This function takes a list of some type, but returns only one of that type, which means that the function is merging or choosing elements from the list. Examples of functions with this signature are `List.sum`, `List.max`, `List.head` and so on.

```
// function signature 6
('a -> bool) -> 'a list -> 'a list
```

This function takes two parameters: the first is a function that maps something to a bool (a predicate), and the second is a list. The return value is a list of the same type. Predicates are used to determine whether a value meets some sort of criteria, so it looks like the function is choosing elements from the list based on whether the predicate is true or not and then returning a subset of the original list. A typical function with this signature is `List.filter`.

```
// function signature 7
('a -> 'b) -> 'a list -> 'b list
```

This function takes two parameters: the first maps type `'a` to type `'b`, and the second is a list of `'a`. The return value is a list of a different type `'b`. A reasonable guess is that the function takes each of the `'a`s in the list, maps them to a `'b` using the function passed in as the first parameter, and returns the new list of `'b`s. And indeed, the prototypical function with this signature is `List.map`.

Using function signatures to find a library method

Function signatures are an important part of searching for library functions. The F# libraries have hundreds of functions in them and they can initially be overwhelming. Unlike an object oriented language, you cannot simply "dot into" an object to find all the appropriate methods. However, if you know the signature of the function you are looking for, you can often narrow down the list of candidates quickly.

For example, let's say you have two lists and you are looking for a function to combine them into one. What would the signature be for this function? It would take two list parameters and return a third, all of the same type, giving the signature:

```
'a list -> 'a list -> 'a list
```

Now go to the MSDN documentation for the F# List module, and scan down the list of functions, looking for something that matches. As it happens, there is only one function with that signature:

```
append : 'T list -> 'T list -> 'T list
```

which is exactly the one we want!

Defining your own types for function signatures

Sometimes you may want to create your own types to match a desired function signature. You can do this using the "type" keyword, and define the type in the same way that a signature is written:

```
type Adder = int -> int
type AdderGenerator = int -> Adder
```

You can then use these types to constrain function values and parameters.

For example, the second definition below will fail because of type constraints. If you remove the type constraint (as in the third definition) there will not be any problem.

```
let a:AdderGenerator = fun x -> (fun y -> x + y)
let b:AdderGenerator = fun (x:float) -> (fun y -> x + y)
let c                = fun (x:float) -> (fun y -> x + y)
```

Test your understanding of function signatures

How well do you understand function signatures? See if you can create simple functions that have each of these signatures. Avoid using explicit type annotations!

```
val testA = int -> int
val testB = int -> int -> int
val testC = int -> (int -> int)
val testD = (int -> int) -> int
val testE = int -> int -> int -> int
val testF = (int -> int) -> (int -> int)
val testG = int -> (int -> int) -> int
val testH = (int -> int -> int) -> int
```

Organizing functions

Now that you know how to define functions, how can you organize them?

In F#, there are three options:

- functions can be nested inside other functions.
- at an application level, the top level functions are grouped into "modules".
- alternatively, you can also use the object-oriented approach and attach functions to types as methods.

We'll look at the first two options in this post, and the third in the next post.

Nested Functions

In F#, you can define functions inside other functions. This is a great way to encapsulate "helper" functions that are needed for the main function but shouldn't be exposed outside.

In the example below `add` is nested inside `addThreeNumbers` :

```
let addThreeNumbers x y z  =

    //create a nested helper function
    let add n =
       fun x -> x + n

    // use the helper function
    x |> add y |> add z

// test
addThreeNumbers 2 3 4
```

A nested function can access its parent function parameters directly, because they are in scope. So, in the example below, the `printError` nested function does not need to have any parameters of its own -- it can access the `n` and `max` values directly.

```
let validateSize max n  =

    //create a nested helper function with no params
    let printError() =
        printfn "Oops: '%i' is bigger than max: '%i'" n max

    // use the helper function
    if n > max then printError()

// test
validateSize 10 9
validateSize 10 11
```

A very common pattern is that the main function defines a nested recursive helper function, and then calls it with the appropriate initial values. The code below is an example of this:

```
let sumNumbersUpTo max =

    // recursive helper function with accumulator
    let rec recursiveSum n sumSoFar =
        match n with
        | 0 -> sumSoFar
        | _ -> recursiveSum (n-1) (n+sumSoFar)

    // call helper function with initial values
    recursiveSum max 0

// test
sumNumbersUpTo 10
```

When nesting functions, do try to avoid very deeply nested functions, especially if the nested functions directly access the variables in their parent scopes rather than having parameters passed to them. A badly nested function will be just as confusing as the worst kind of deeply nested imperative branching.

Here's how *not* to do it:

```
// wtf does this function do?
let f x =
    let f2 y =
        let f3 z =
            x * z
        let f4 z =
            let f5 z =
                y * z
            let f6 () =
                y * x
            f6()
        f4 y
    x * f2 x
```

Modules

A module is just a set of functions that are grouped together, typically because they work on the same data type or types.

A module definition looks very like a function definition. It starts with the `module` keyword, then an `=` sign, and then the contents of the module are listed. The contents of the module *must* be indented, just as expressions in a function definition must be indented.

Here's a module that contains two functions:

```
module MathStuff =

    let add x y  = x + y
    let subtract x y  = x - y
```

Now if you try this in Visual Studio, and you hover over the `add` function, you will see that the full name of the `add` function is actually `MathStuff.add`, just as if `MathStuff` was a class and `add` was a method.

Actually, that's exactly what is going on. Behind the scenes, the F# compiler creates a static class with static methods. So the C# equivalent would be:

```
static class MathStuff
{
    static public int add(int x, int y)
    {
        return x + y;
    }

    static public int subtract(int x, int y)
    {
        return x - y;
    }
}
```

If you realize that modules are just static classes, and that functions are static methods, then you will already have a head-start on understanding how modules work in F#, as most of the rules that apply to static classes also apply to modules.

And, just as in C# every standalone function must be part of a class, in F# every standalone function *must* be part of a module.

Accessing functions across module boundaries

If you want to access a function in another module, you can refer to it by its qualified name.

```
module MathStuff =

    let add x y  = x + y
    let subtract x y  = x - y

module OtherStuff =

    // use a function from the MathStuff module
    let add1 x = MathStuff.add x 1
```

You can also import all the functions in another module with the `open` directive, after which you can use the short name, rather than having to specify the qualified name.

```
module OtherStuff =
    open MathStuff  // make all functions accessible

    let add1 x = add x 1
```

The rules for using qualified names are exactly as you would expect. That is, you can always use a fully qualified name to access a function, and you can use relative names or unqualified names based on what other modules are in scope.

Nested modules

Just like static classes, modules can contain child modules nested within them, as shown below:

```
module MathStuff =

    let add x y  = x + y
    let subtract x y  = x - y

    // nested module
    module FloatLib =

        let add x y :float = x + y
        let subtract x y :float  = x - y
```

And other modules can reference functions in the nested modules using either a full name or a relative name as appropriate:

```
module OtherStuff =
    open MathStuff

    let add1 x = add x 1

    // fully qualified
    let add1Float x = MathStuff.FloatLib.add x 1.0

    //with a relative path
    let sub1Float x = FloatLib.subtract x 1.0
```

Top level modules

So if there can be nested child modules, that implies that, going back up the chain, there must always be some *top-level* parent module. This is indeed true.

Top level modules are defined slightly differently than the modules we have seen so far.

- The `module MyModuleName` line *must* be the first declaration in the file
- There is no `=` sign
- The contents of the module are *not* indented

In general, there must be a top level module declaration present in every `.FS` source file. There some exceptions, but it is good practice anyway. The module name does not have to be the same as the name of the file, but two files cannot share the same module name.

For `.FSX` script files, the module declaration is not needed, in which case the module name is automatically set to the filename of the script.

Here is an example of `MathStuff` declared as a top level module:

```
// top level module
module MathStuff

let add x y  = x + y
let subtract x y  = x - y

// nested module
module FloatLib =

    let add x y :float = x + y
    let subtract x y :float  = x - y
```

Note the lack of indentation for the top level code (the contents of `module MathStuff`), but that the content of a nested module like `FloatLib` does still need to be indented.

Other module content

A module can contain other declarations as well as functions, including type declarations, simple values and initialization code (like static constructors)

```
module MathStuff =

    // functions
    let add x y  = x + y
    let subtract x y  = x - y

    // type definitions
    type Complex = {r:float; i:float}
    type IntegerFunction = int -> int -> int
    type DegreesOrRadians = Deg | Rad

    // "constant"
    let PI = 3.141

    // "variable"
    let mutable TrigType = Deg

    // initialization / static constructor
    do printfn "module initialized"
```

By the way, if you are playing with these examples in the interactive window, you might want to right-click and do "Reset Session" every so often, so that the code is fresh and

doesn't get contaminated with previous evaluations

Shadowing

Here's our example module again. Notice that `MathStuff` has an `add` function and `FloatLib` *also* has an `add` function.

```
module MathStuff =

    let add x y  = x + y
    let subtract x y  = x - y

    // nested module
    module FloatLib =

        let add x y :float = x + y
        let subtract x y :float  = x - y
```

Now what happens if I bring *both* of them into scope, and then use `add` ?

```
open  MathStuff
open  MathStuff.FloatLib

let result = add 1 2  // Compiler error: This expression was expected to
                      // have type float but here has type int
```

What happened was that the `MathStuff.FloatLib` module has masked or overridden the original `MathStuff` module, which has been "shadowed" by `FloatLib` .

As a result you now get a FS0001 compiler error because the first parameter `1` is expected to be a float. You would have to change `1` to `1.0` to fix this.

Unfortunately, this is invisible and easy to overlook. Sometimes you can do cool tricks with this, almost like subclassing, but more often, it can be annoying if you have functions with the same name (such as the very common `map`).

If you don't want this to happen, there is a way to stop it by using the `RequireQualifiedAccess` attribute. Here's the same example where both modules are decorated with it.

```
[<RequireQualifiedAccess>]
module MathStuff =

    let add x y  = x + y
    let subtract x y  = x - y

    // nested module
    [<RequireQualifiedAccess>]
    module FloatLib =

        let add x y :float = x + y
        let subtract x y :float  = x - y
```

Now the `open` isn't allowed:

```
open  MathStuff   // error
open  MathStuff.FloatLib // error
```

But we can still access the functions (without any ambiguity) via their qualified name:

```
let result = MathStuff.add 1 2
let result = MathStuff.FloatLib.add 1.0 2.0
```

Access Control

F# supports the use of standard .NET access control keywords such as `public`, `private`, and `internal`. The MSDN documentation has the complete details.

- These access specifiers can be put on the top-level ("let bound") functions, values, types and other declarations in a module. They can also be specified for the modules themselves (you might want a private nested module, for example).
- Everything is public by default (with a few exceptions) so you will need to use `private` or `internal` if you want to protect them.

These access specifiers are just one way of doing access control in F#. Another completely different way is to use module "signature" files, which are a bit like C header files. They describe the content of the module in an abstract way. Signatures are very useful for doing serious encapsulation, but that discussion will have to wait for the planned series on encapsulation and capability based security.

Namespaces

Namespaces in F# are similar to namespaces in C#. They can be used to organize modules and types to avoid name collisions.

A namespace is declared with a `namespace` keyword, as shown below.

```
namespace Utilities

module MathStuff =

    // functions
    let add x y  = x + y
    let subtract x y  = x - y
```

Because of this namespace, the fully qualified name of the `MathStuff` module now becomes `Utilities.MathStuff` and the fully qualified name of the `add` function now becomes `Utilities.MathStuff.add`.

With the namespace, the indentation rules apply, so that the module defined above must have its content indented, as it it were a nested module.

You can also declare a namespace implicitly by adding dots to the module name. That is, the code above could also be written as:

```
module Utilities.MathStuff

// functions
let add x y  = x + y
let subtract x y  = x - y
```

The fully qualified name of the `MathStuff` module is still `Utilities.MathStuff`, but in this case, the module is a top-level module and the contents do not need to be indented.

Some additional things to be aware of when using namespaces:

- Namespaces are optional for modules. And unlike C#, there is no default namespace for an F# project, so a top level module without a namespace will be at the global level. If you are planning to create reusable libraries, be sure to add some sort of namespace to avoid naming collisions with code in other libraries.
- Namespaces can directly contain type declarations, but not function declarations. As noted earlier, all function and value declarations must be part of a module.
- Finally, be aware that namespaces don't work well in scripts. For example, if you try to to send a namespace declaration such as `namespace Utilities` below to the interactive window, you will get an error.

Namespace hierarchies

You can create a namespace hierarchy by simply separating the names with periods:

```
namespace Core.Utilities

module MathStuff =
    let add x y  = x + y
```

And if you want to put *two* namespaces in the same file, you can. Note that all namespaces *must* be fully qualified -- there is no nesting.

```
namespace Core.Utilities

module MathStuff =
    let add x y  = x + y

namespace Core.Extra

module MoreMathStuff =
    let add x y  = x + y
```

One thing you can't do is have a naming collision between a namespace and a module.

```
namespace Core.Utilities

module MathStuff =
    let add x y  = x + y

namespace Core

// fully qualified name of module
// is Core.Utilities
// Collision with namespace above!
module Utilities =
    let add x y  = x + y
```

Mixing types and functions in modules

We've seen that a module typically consists of a set of related functions that act on a data type.

In an object oriented program, the data structure and the functions that act on it would be combined in a class. However in functional-style F#, a data structure and the functions that act on it are combined in a module instead.

There are two common patterns for mixing types and functions together:

- having the type declared separately from the functions
- having the type declared in the same module as the functions

In the first approach, the type is declared *outside* any module (but in a namespace) and then the functions that work on the type are put in a module with a similar name.

```
// top-level module
namespace Example

// declare the type outside the module
type PersonType = {First:string; Last:string}

// declare a module for functions that work on the type
module Person =

    // constructor
    let create first last =
        {First=first; Last=last}

    // method that works on the type
    let fullName {First=first; Last=last} =
        first + " " + last

// test
let person = Person.create "john" "doe"
Person.fullName person |> printfn "Fullname=%s"
```

In the alternative approach, the type is declared *inside* the module and given a simple name such as " `T` " or the name of the module. So the functions are accessed with names like `MyModule.Func1` and `MyModule.Func2` while the type itself is accessed with a name like `MyModule.T` . Here's an example:

```
module Customer =

    // Customer.T is the primary type for this module
    type T = {AccountId:int; Name:string}

    // constructor
    let create id name =
        {T.AccountId=id; T.Name=name}

    // method that works on the type
    let isValid {T.AccountId=id; } =
        id > 0

// test
let customer = Customer.create 42 "bob"
Customer.isValid customer |> printfn "Is valid?=%b"
```

Note that in both cases, you should have a constructor function that creates new instances of the type (a factory method, if you will), Doing this means that you will rarely have to explicitly name the type in your client code, and therefore, you not should not care whether it lives in the module or not!

So which approach should you choose?

- The former approach is more .NET like, and much better if you want to share your libraries with other non-F# code, as the exported class names are what you would expect.
- The latter approach is more common for those used to other functional languages. The type inside a module compiles into nested classes, which is not so nice for interop.

For yourself, you might want to experiment with both. And in a team programming situation, you should choose one style and be consistent.

Modules containing types only

If you have a set of types that you need to declare without any associated functions, don't bother to use a module. You can declare types directly in a namespace and avoid nested classes.

For example, here is how you might think to do it:

```
// top-level module
module Example

// declare the type inside a module
type PersonType = {First:string; Last:string}

// no functions in the module, just types...
```

And here is a alternative way to do it. The `module` keyword has simply been replaced with `namespace`.

```
// use a namespace
namespace Example

// declare the type outside any module
type PersonType = {First:string; Last:string}
```

In both cases, `PersonType` will have the same fully qualified name.

Note that this only works with types. Functions must always live in a module.

Attaching functions to types

Although we have focused on the pure functional style so far, sometimes it is convenient to switch to an object oriented style. And one of the key features of the OO style is the ability to attach functions to a class and "dot into" the class to get the desired behavior.

In F#, this is done using a feature called "type extensions". And any F# type, not just classes, can have functions attached to them.

Here's an example of attaching a function to a record type.

```
module Person =
    type T = {First:string; Last:string} with
        // member defined with type declaration
        member this.FullName =
            this.First + " " + this.Last

    // constructor
    let create first last =
        {First=first; Last=last}

// test
let person = Person.create "John" "Doe"
let fullname = person.FullName
```

The key things to note are:

- The `with` keyword indicates the start of the list of members
- The `member` keyword shows that this is a member function (i.e. a method)
- The word `this` is a placeholder for the object that is being dotted into (called a "self-identifier"). The placeholder prefixes the function name, and then the function body then uses the same placeholder when it needs to refer to the current instance. There is no requirement to use a particular word, just as long as it is consistent. You could use `this` or `self` or `me` or any other word that commonly indicates a self reference.

You don't have to add a member at the same time that you declare the type, you can always add it later in the same module:

```
module Person =
    type T = {First:string; Last:string} with
       // member defined with type declaration
        member this.FullName =
            this.First + " " + this.Last

    // constructor
    let create first last =
        {First=first; Last=last}

    // another member added later
    type T with
        member this.SortableName =
            this.Last + ", " + this.First
// test
let person = Person.create "John" "Doe"
let fullname = person.FullName
let sortableName = person.SortableName
```

These examples demonstrate what are called "intrinsic extensions". They are compiled into the type itself and are always available whenever the type is used. They also show up when you use reflection.

With intrinsic extensions, it is even possible to have a type definition that divided across several files, as long as all the components use the same namespace and are all compiled into the same assembly. Just as with partial classes in C#, this can be useful to separate generated code from authored code.

Optional extensions

Another alternative is that you can add an extra member from a completely different module. These are called "optional extensions". They are not compiled into the type itself, and require some other module to be in scope for them to work (this behavior is just like C# extension methods).

For example, let's say we have a `Person` type defined:

```
module Person =
    type T = {First:string; Last:string} with
       // member defined with type declaration
        member this.FullName =
            this.First + " " + this.Last

    // constructor
    let create first last =
        {First=first; Last=last}

    // another member added later
    type T with
        member this.SortableName =
            this.Last + ", " + this.First
```

The example below demonstrates how to add an `UppercaseName` extension to it in a different module:

```
// in a different module
module PersonExtensions =

    type Person.T with
    member this.UppercaseName =
        this.FullName.ToUpper()
```

So now let's test this extension:

```
let person = Person.create "John" "Doe"
let uppercaseName = person.UppercaseName
```

Uh-oh, we have an error. What's wrong is that the `PersonExtensions` is not in scope. Just as for C#, any extensions have to be brought into scope in order to be used.

Once we do that, everything is fine:

```
// bring the extension into scope first!
open PersonExtensions

let person = Person.create "John" "Doe"
let uppercaseName = person.UppercaseName
```

Extending system types

You can extend types that are in the .NET libraries as well. But be aware that when extending a type, you must use the actual type name, not a type abbreviation.

For example, if you try to extend `int`, you will fail, because `int` is not the true name of the type:

```
type int with
    member this.IsEven = this % 2 = 0
```

You must use `System.Int32` instead:

```
type System.Int32 with
    member this.IsEven = this % 2 = 0

//test
let i = 20
if i.IsEven then printfn "'%i' is even" i
```

Static members

You can make the member functions static by:

- adding the keyword `static`
- dropping the `this` placeholder

```
module Person =
    type T = {First:string; Last:string} with
        // member defined with type declaration
        member this.FullName =
            this.First + " " + this.Last

        // static constructor
        static member Create first last =
            {First=first; Last=last}

// test
let person = Person.T.Create "John" "Doe"
let fullname = person.FullName
```

And you can create static members for system types as well:

```
type System.Int32 with
    static member IsOdd x = x % 2 = 1

type System.Double with
    static member Pi = 3.141

//test
let result = System.Int32.IsOdd 20
let pi = System.Double.Pi
```

Attaching existing functions

A very common pattern is to attach pre-existing standalone functions to a type. This has a couple of benefits:

- While developing, you can create standalone functions that refer to other standalone functions. This makes programming easier because type inference works much better with functional-style code than with OO-style ("dotting into") code.
- But for certain key functions, you can attach them to the type as well. This gives clients the choice of whether to use functional or object-oriented style.

One example of this in the F# libraries is the function that calculates a list's length. It is available as a standalone function in the `List` module, but also as a method on a list instance.

```
let list = [1..10]

// functional style
let len1 = List.length list

// OO style
let len2 = list.Length
```

In the following example, we start with a type with no members initially, then define some functions, then finally attach the `fullName` function to the type.

```
module Person =
    // type with no members initially
    type T = {First:string; Last:string}

    // constructor
    let create first last =
        {First=first; Last=last}

    // standalone function
    let fullName {First=first; Last=last} =
        first + " " + last

    // attach preexisting function as a member
    type T with
        member this.FullName = fullName this

// test
let person = Person.create "John" "Doe"
let fullname = Person.fullName person  // functional style
let fullname2 = person.FullName        // OO style
```

The standalone `fullName` function has one parameter, the person. In the attached member, the parameter comes from the `this` self-reference.

Attaching existing functions with multiple parameters

One nice thing is that when the previously defined function has multiple parameters, you don't have to respecify them all when doing the attachment, as long as the `this` parameter is first.

In the example below, the `hasSameFirstAndLastName` function has three parameters. Yet when we attach it, we only need to specify one!

```
module Person =
    // type with no members initially
    type T = {First:string; Last:string}

    // constructor
    let create first last =
        {First=first; Last=last}

    // standalone function
    let hasSameFirstAndLastName (person:T) otherFirst otherLast =
        person.First = otherFirst && person.Last = otherLast

    // attach preexisting function as a member
    type T with
        member this.HasSameFirstAndLastName = hasSameFirstAndLastName this

// test
let person = Person.create "John" "Doe"
let result1 = Person.hasSameFirstAndLastName person "bob" "smith" // functional style
let result2 = person.HasSameFirstAndLastName "bob" "smith" // OO style
```

Why does this work? Hint: think about currying and partial application!

Tuple-form methods

When we start having methods with more than one parameter, we have to make a decision:

- we could use the standard (curried) form, where parameters are separated with spaces, and partial application is supported.
- we could pass in *all* the parameters at once, comma-separated, in a single tuple.

The "curried" form is more functional, and the "tuple" form is more object-oriented.

The tuple form is also how F# interacts with the standard .NET libraries, so let's examine this approach in more detail.

As a testbed, here is a Product type with two methods, each implemented using one of the approaches. The `CurriedTotal` and `TupleTotal` methods each do the same thing: work out the total price for a given quantity and discount.

```
type Product = {SKU:string; Price: float} with

    // curried style
    member this.CurriedTotal qty discount =
        (this.Price * float qty) - discount

    // tuple style
    member this.TupleTotal(qty,discount) =
        (this.Price * float qty) - discount
```

And here's some test code:

```
let product = {SKU="ABC"; Price=2.0}
let total1 = product.CurriedTotal 10 1.0
let total2 = product.TupleTotal(10,1.0)
```

No difference so far.

We know that curried version can be partially applied:

```
let totalFor10 = product.CurriedTotal 10
let discounts = [1.0..5.0]
let totalForDifferentDiscounts
    = discounts |> List.map totalFor10
```

But the tuple approach can do a few things that that the curried one can't, namely:

- Named parameters
- Optional parameters
- Overloading

Named parameters with tuple-style parameters

The tuple-style approach supports named parameters:

```
let product = {SKU="ABC"; Price=2.0}
let total3 = product.TupleTotal(qty=10,discount=1.0)
let total4 = product.TupleTotal(discount=1.0, qty=10)
```

As you can see, when names are used, the parameter order can be changed.

Note: if some parameters are named and some are not, the named ones must always be last.

Optional parameters with tuple-style parameters

For tuple-style methods, you can specify an optional parameter by prefixing the parameter name with a question mark.

- If the parameter is set, it comes through as `Some value`
- If the parameter is not set, it comes through as `None`

Here's an example:

```
type Product = {SKU:string; Price: float} with

    // optional discount
    member this.TupleTotal2(qty,?discount) =
        let extPrice = this.Price * float qty
        match discount with
        | None -> extPrice
        | Some discount -> extPrice - discount
```

And here's a test:

```
let product = {SKU="ABC"; Price=2.0}

// discount not specified
let total1 = product.TupleTotal2(10)

// discount specified
let total2 = product.TupleTotal2(10,1.0)
```

This explicit matching of the `None` and `Some` can be tedious, and there is a slightly more elegant solution for handling optional parameters.

There is a function `defaultArg` which takes the parameter as the first argument and a default for the second argument. If the parameter is set, the value is returned. And if not, the default value is returned.

Let's see the same code rewritten to use `defaultArg`

```
type Product = {SKU:string; Price: float} with

    // optional discount
    member this.TupleTotal2(qty,?discount) =
        let extPrice = this.Price * float qty
        let discount = defaultArg discount 0.0
        //return
        extPrice - discount
```

Method overloading

In C#, you can have multiple methods with the same name that differ only in their function signature (e.g. different parameter types and/or number of parameters)

In the pure functional model, that does not make sense -- a function works with a particular domain type and a particular range type. The same function cannot work with different domains and ranges.

However, F# *does* support method overloading, but only for methods (that is functions attached to types) and of these, only those using tuple-style parameter passing.

Here's an example, with yet another variant on the `TupleTotal` method!

```
type Product = {SKU:string; Price: float} with

    // no discount
    member this.TupleTotal3(qty) =
        printfn "using non-discount method"
        this.Price * float qty

    // with discount
    member this.TupleTotal3(qty, discount) =
        printfn "using discount method"
        (this.Price * float qty) - discount
```

Normally, the F# compiler would complain that there are two methods with the same name, but in this case, because they are tuple based and because their signatures are different, it is acceptable. (To make it obvious which one is being called, I have added a small debugging message.)

And here's a test:

```
let product = {SKU="ABC"; Price=2.0}

// discount not specified
let total1 = product.TupleTotal3(10)

// discount specified
let total2 = product.TupleTotal3(10,1.0)
```

Hey! Not so fast... The downsides of using methods

If you are coming from an object-oriented background, you might be tempted to use methods everywhere, because that is what you are familiar with. But be aware that there some major downsides to using methods as well:

- Methods don't play well with type inference
- Methods don't play well with higher order functions

In fact, by overusing methods you would be needlessly bypassing the most powerful and useful aspects of programming in F#.

Let's see what I mean.

Methods don't play well with type inference

Let's go back to our Person example again, the one that had the same logic implemented both as a standalone function and as a method:

```
module Person =
    // type with no members initially
    type T = {First:string; Last:string}

    // constructor
    let create first last =
        {First=first; Last=last}

    // standalone function
    let fullName {First=first; Last=last} =
        first + " " + last

    // function as a member
    type T with
        member this.FullName = fullName this
```

Now let's see how well each one works with type inference. Say that I want to print the full name of a person, so I will define a function `printFullName` that takes a person as a parameter.

Here's the code using the module level standalone function.

```
open Person

// using standalone function
let printFullName person =
    printfn "Name is %s" (fullName person)

// type inference worked:
//    val printFullName : Person.T -> unit
```

This compiles without problems, and the type inference has correctly deduced that parameter was a person

Now let's try the "dotted" version:

```
open Person

// using method with "dotting into"
let printFullName2 person =
    printfn "Name is %s" (person.FullName)
```

This does not compile at all, because the type inference does not have enough information to deduce the parameter. *Any* object might implement `.FullName` -- there is just not enough to go on.

Yes, we could annotate the function with the parameter type, but that defeats the whole purpose of type inference.

Methods don't play well with higher order functions

A similar problem happens with higher order functions. For example, let's say that, given a list of people, we want to get all their full names.

With a standalone function, this is trivial:

```
open Person

let list = [
    Person.create "Andy" "Anderson";
    Person.create "John" "Johnson";
    Person.create "Jack" "Jackson"]

//get all the full names at once
list |> List.map fullName
```

With object methods, we have to create special lambdas everywhere:

```
open Person

let list = [
    Person.create "Andy" "Anderson";
    Person.create "John" "Johnson";
    Person.create "Jack" "Jackson"]

//get all the full names at once
list |> List.map (fun p -> p.FullName)
```

And this is just a simple example. Object methods don't compose well, are hard to pipe, and so on.

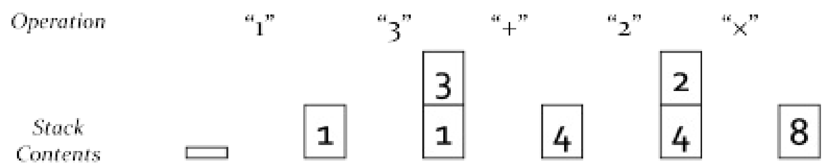
So, a plea for those of you new to functionally programming. Don't use methods at all if you can, especially when you are learning. They are a crutch that will stop you getting the full benefit from functional programming.

Worked example: A stack based calculator

In this post, we'll implement a simple stack based calculator (also known as "reverse Polish" style). The implementation is almost entirely done with functions, with only one special type and no pattern matching at all, so it is a great testing ground for the concepts introduced in this series.

If you are not familiar with a stack based calculator, it works as follows: numbers are pushed on a stack, and operations such as addition and multiplication pop numbers off the stack and push the result back on.

Here is a diagram showing a simple calculation using a stack:



The first steps to designing a system like this is to think about how it would be used. Following a Forth like syntax, we will give each action a label, so that the example above might want to be written something like:

```
EMPTY ONE THREE ADD TWO MUL SHOW
```

We might not be able to get this exact syntax, but let's see how close we can get.

The Stack data type

First we need to define the data structure for a stack. To keep things simple, we'll just use a list of floats.

```
type Stack = float list
```

But, hold on, let's wrap it in a single case union type to make it more descriptive, like this:

```
type Stack = StackContents of float list
```

For more details on why this is nicer, read the discussion of single case union types in this post.

Now, to create a new stack, we use `StackContents` as a constructor:

```
let newStack = StackContents [1.0;2.0;3.0]
```

And to extract the contents of an existing Stack, we pattern match with `StackContents` :

```
let (StackContents contents) = newStack

// "contents" value set to
// float list = [1.0; 2.0; 3.0]
```

The Push function

Next we need a way to push numbers on to the stack. This will be simply be prepending the new value at the front of the list using the " `::` " operator.

Here is our push function:

```
let push x aStack =
    let (StackContents contents) = aStack
    let newContents = x::contents
    StackContents newContents
```

This basic function has a number of things worth discussing.

First, note that the list structure is immutable, so the function must accept an existing stack and return a new stack. It cannot just alter the existing stack. In fact, all of the functions in this example will have a similar format like this:

```
Input: a Stack plus other parameters
Output: a new Stack
```

Next, what should the order of the parameters be? Should the stack parameter come first or last? If you remember the discussion of designing functions for partial application, you will remember that the most changeable thing should come last. You'll see shortly that this guideline will be born out.

Finally, the function can be made more concise by using pattern matching in the function parameter itself, rather than using a `let` in the body of the function.

Here is the rewritten version:

```
let push x (StackContents contents) =
    StackContents (x::contents)
```

Much nicer!

And by the way, look at the nice signature it has:

```
val push : float -> Stack -> Stack
```

As we know from a previous post, the signature tells you a lot about the function. In this case, I could probably guess what it did from the signature alone, even without knowing that the name of the function was "push". This is one of the reasons why it is a good idea to have explicit type names. If the stack type had just been a list of floats, it wouldn't have been as self-documenting.

Anyway, now let's test it:

```
let emptyStack = StackContents []
let stackWith1 = push 1.0 emptyStack
let stackWith2 = push 2.0 stackWith1
```

Works great!

Building on top of "push"

With this simple function in place, we can easily define an operation that pushes a particular number onto the stack.

```
let ONE stack = push 1.0 stack
let TWO stack = push 2.0 stack
```

But wait a minute! Can you see that the `stack` parameter is used on both sides? In fact, we don't need to mention it at all. Instead we can skip the `stack` parameter and write the functions using partial application as follows:

```
let ONE = push 1.0
let TWO = push 2.0
let THREE = push 3.0
let FOUR = push 4.0
let FIVE = push 5.0
```

Now you can see that if the parameters for `push` were in a different order, we wouldn't have been able to do this.

While we're at it, let's define a function that creates an empty stack as well:

```
let EMPTY = StackContents []
```

Let's test all of these now:

```
let stackWith1 = ONE EMPTY
let stackWith2 = TWO stackWith1
let stackWith3  = THREE stackWith2
```

These intermediate stacks are annoying ? can we get rid of them? Yes! Note that these functions ONE, TWO, THREE all have the same signature:

```
Stack -> Stack
```

This means that they can be chained together nicely! The output of one can be fed into the input of the next, as shown below:

```
let result123 = EMPTY |> ONE |> TWO |> THREE
let result312 = EMPTY |> THREE |> ONE |> TWO
```

Popping the stack

That takes care of pushing onto the stack ? what about a `pop` function next?

When we pop the stack, we will return the top of the stack, obviously, but is that all?

In an object-oriented style, the answer is yes. In an OO approach, we would *mutate* the stack itself behind the scenes, so that the top element was removed.

But in a functional style, the stack is immutable. The only way to remove the top element is to create a *new stack* with the element removed. In order for the caller to have access to this new diminished stack, it needs to be returned along with the top element itself.

In other words, the `pop` function will have to return *two* values, the top plus the new stack. The easiest way to do this in F# is just to use a tuple.

Here's the implementation:

```
/// Pop a value from the stack and return it
/// and the new stack as a tuple
let pop (StackContents contents) =
    match contents with
    | top::rest ->
        let newStack = StackContents rest
        (top,newStack)
```

This function is also very straightforward.

As before, we are extracting the `contents` directly in the parameter.

We then use a `match..with` expression to test the contents.

Next, we separate the top element from the rest, create a new stack from the remaining elements and finally return the pair as a tuple.

Try the code above and see what happens. You will get a compiler error! The compiler has caught a case we have overlooked -- what happens if the stack is empty?

So now we have to decide how to handle this.

- Option 1: Return a special "Success" or "Error" state, as we did in a post from the "why use F#?" series.
- Option 2: Throw an exception.

Generally, I prefer to use error cases, but in this case, we'll use an exception. So here's the `pop` code changed to handle the empty case:

```
/// Pop a value from the stack and return it
/// and the new stack as a tuple
let pop (StackContents contents) =
    match contents with
    | top::rest ->
        let newStack = StackContents rest
        (top,newStack)
    | [] ->
        failwith "Stack underflow"
```

Now let's test it:

```
let initialStack = EMPTY  |> ONE |> TWO
let popped1, poppedStack = pop initialStack
let popped2, poppedStack2 = pop poppedStack
```

and to test the underflow:

```
let _ = pop EMPTY
```

Writing the math functions

Now with both push and pop in place, we can work on the "add" and "multiply" functions:

```
let ADD stack =
   let x,s = pop stack  //pop the top of the stack
   let y,s2 = pop s     //pop the result stack
   let result = x + y   //do the math
   push result s2       //push back on the doubly-popped stack

let MUL stack =
   let x,s = pop stack  //pop the top of the stack
   let y,s2 = pop s     //pop the result stack
   let result = x * y   //do the math
   push result s2       //push back on the doubly-popped stack
```

Test these interactively:

```
let add1and2 = EMPTY |> ONE |> TWO |> ADD
let add2and3 = EMPTY |> TWO |> THREE |> ADD
let mult2and3 = EMPTY |> TWO |> THREE |> MUL
```

It works!

Time to refactor...

It is obvious that there is significant duplicate code between these two functions. How can we refactor?

Both functions pop two values from the stack, apply some sort of binary function, and then push the result back on the stack. This leads us to refactor out the common code into a "binary" function that takes a two parameter math function as a parameter:

```
let binary mathFn stack =
    // pop the top of the stack
    let y,stack' = pop stack
    // pop the top of the stack again
    let x,stack'' = pop stack'
    // do the math
    let z = mathFn x y
    // push the result value back on the doubly-popped stack
    push z stack''
```

Note that in this implementation, I've switched to using ticks to represent changed states of the "same" object, rather than numeric suffixes. Numeric suffixes can easily get quite confusing.

Question: why are the parameters in the order they are, instead of `mathFn` being after `stack`?

Now that we have `binary`, we can define ADD and friends more simply:

Here's a first attempt at ADD using the new `binary` helper:

```
let ADD aStack = binary (fun x y -> x + y) aStack
```

But we can eliminate the lambda, as it is *exactly* the definition of the built-in `+` function! Which gives us:

```
let ADD aStack = binary (+) aStack
```

And again, we can use partial application to hide the stack parameter. Here's the final definition:

```
let ADD = binary (+)
```

And here's the definition of some other math functions:

```
let SUB = binary (-)
let MUL = binary (*)
let DIV = binary (../)
```

Let's test interactively again.

```
let div2by3 = EMPTY |> THREE|> TWO |> DIV
let sub2from5 = EMPTY  |> TWO |> FIVE |> SUB
let add1and2thenSub3 = EMPTY |> ONE |> TWO |> ADD |> THREE |> SUB
```

In a similar fashion, we can create a helper function for unary functions

```
let unary f stack =
    let x,stack' = pop stack  //pop the top of the stack
    push (f x) stack'         //push the function value on the stack
```

And then define some unary functions:

```
let NEG = unary (fun x -> -x)
let SQUARE = unary (fun x -> x * x)
```

Test interactively again:

```
let neg3 = EMPTY  |> THREE|> NEG
let square2 = EMPTY  |> TWO |> SQUARE
```

Putting it all together

In the original requirements, we mentioned that we wanted to be able to show the results, so let's define a SHOW function.

```
let SHOW stack =
    let x,_ = pop stack
    printfn "The answer is %f" x
    stack  // keep going with same stack
```

Note that in this case, we pop the original stack but ignore the diminished version. The final result of the function is the original stack, as if it had never been popped.

So now finally, we can write the code example from the original requirements

```
EMPTY |> ONE |> THREE |> ADD |> TWO |> MUL |> SHOW
```

Going further

This is fun -- what else can we do?

Well, we can define a few more core helper functions:

```
/// Duplicate the top value on the stack
let DUP stack =
    // get the top of the stack
    let x,_ = pop stack
    // push it onto the stack again
    push x stack

/// Swap the top two values
let SWAP stack =
    let x,s = pop stack
    let y,s' = pop s
    push y (push x s')

/// Make an obvious starting point
let START  = EMPTY
```

And with these additional functions in place, we can write some nice examples:

```
START
    |> ONE |> TWO |> SHOW

START
    |> ONE |> TWO |> ADD |> SHOW
    |> THREE |> ADD |> SHOW

START
    |> THREE |> DUP |> DUP |> MUL |> MUL // 27

START
    |> ONE |> TWO |> ADD |> SHOW  // 3
    |> THREE |> MUL |> SHOW       // 9
    |> TWO |> SWAP |> DIV |> SHOW // 9 div 2 = 4.5
```

Using composition instead of piping

But that's not all. In fact, there is another very interesting way to think about these functions.

As I pointed out earlier, they all have an identical signature:

```
Stack -> Stack
```

So, because the input and output types are the same, these functions can be composed using the composition operator `>>`, not just chained together with pipes.

Here are some examples:

```
// define a new function
let ONE_TWO_ADD =
    ONE >> TWO >> ADD

// test it
START |> ONE_TWO_ADD |> SHOW

// define a new function
let SQUARE =
    DUP >> MUL

// test it
START |> TWO |> SQUARE |> SHOW

// define a new function
let CUBE =
    DUP >> DUP >> MUL >> MUL

// test it
START |> THREE |> CUBE |> SHOW

// define a new function
let SUM_NUMBERS_UPTO =
    DUP                      // n
    >> ONE >> ADD            // n+1
    >> MUL                   // n(n+1)
    >> TWO >> SWAP >> DIV    // n(n+1) / 2

// test it
START |> THREE |> SQUARE |> SUM_NUMBERS_UPTO |> SHOW
```

In each of these cases, a new function is defined by composing other functions together to make a new one. This is a good example of the "combinator" approach to building up functionality.

Pipes vs composition

We have now seen two different ways that this stack based model can be used; by piping or by composition. So what is the difference? And why would we prefer one way over another?

The difference is that piping is, in a sense, a "realtime transformation" operation. When you use piping you are actually doing the operations right now, passing a particular stack around.

On the other hand, composition is a kind of "plan" for what you want to do, building an overall function from a set of parts, but *not* actually running it yet.

So for example, I can create a "plan" for how to square a number by combining smaller operations:

```
let COMPOSED_SQUARE = DUP >> MUL
```

I cannot do the equivalent with the piping approach.

```
let PIPED_SQUARE = DUP |> MUL
```

This causes a compilation error. I have to have some sort of concrete stack instance to make it work:

```
let stackWith2 = EMPTY |> TWO
let twoSquared = stackWith2 |> DUP |> MUL
```

And even then, I only get the answer for this particular input, not a plan for all possible inputs, as in the COMPOSED_SQUARE example.

The other way to create a "plan" is to explicitly pass in a lambda to a more primitive function, as we saw near the beginning:

```
let LAMBDA_SQUARE = unary (fun x -> x * x)
```

This is much more explicit (and is likely to be faster) but loses all the benefits and clarity of the composition approach.

So, in general, go for the composition approach if you can!

The complete code

Here's the complete code for all the examples so far.

```
// ==============================================
// Types
// ==============================================

type Stack = StackContents of float list

// ==============================================
// Stack primitives
// ==============================================

/// Push a value on the stack
```

```
let push x (StackContents contents) =
    StackContents (x::contents)

/// Pop a value from the stack and return it
/// and the new stack as a tuple
let pop (StackContents contents) =
    match contents with
    | top::rest ->
        let newStack = StackContents rest
        (top,newStack)
    | [] ->
        failwith "Stack underflow"

// ==============================================
// Operator core
// ==============================================

// pop the top two elements
// do a binary operation on them
// push the result
let binary mathFn stack =
    let y,stack' = pop stack
    let x,stack'' = pop stack'
    let z = mathFn x y
    push z stack''

// pop the top element
// do a unary operation on it
// push the result
let unary f stack =
    let x,stack' = pop stack
    push (f x) stack'

// ==============================================
// Other core
// ==============================================

/// Pop and show the top value on the stack
let SHOW stack =
    let x,_ = pop stack
    printfn "The answer is %f" x
    stack  // keep going with same stack

/// Duplicate the top value on the stack
let DUP stack =
    let x,s = pop stack
    push x (push x s)

/// Swap the top two values
let SWAP stack =
    let x,s = pop stack
    let y,s' = pop s
    push y (push x s')
```

```
/// Drop the top value on the stack
let DROP stack =
    let _,s = pop stack  //pop the top of the stack
    s                    //return the rest

// ==============================================
// Words based on primitives
// ==============================================

// Constants
// -------------------------------
let EMPTY = StackContents []
let START  = EMPTY


// Numbers
// -------------------------------
let ONE = push 1.0
let TWO = push 2.0
let THREE = push 3.0
let FOUR = push 4.0
let FIVE = push 5.0

// Math functions
// -------------------------------
let ADD = binary (+)
let SUB = binary (-)
let MUL = binary (*)
let DIV = binary (../)

let NEG = unary (fun x -> -x)


// ==============================================
// Words based on composition
// ==============================================

let SQUARE =
    DUP >> MUL

let CUBE =
    DUP >> DUP >> MUL >> MUL

let SUM_NUMBERS_UPTO =
    DUP                      // n
    >> ONE >> ADD            // n+1
    >> MUL                   // n(n+1)
    >> TWO >> SWAP >> DIV    // n(n+1) / 2
```

Summary

So there we have it, a simple stack based calculator. We've seen how we can start with a few primitive operations (`push` , `pop` , `binary` , `unary`) and from them, build up a whole domain specific language that is both easy to implement and easy to use.

As you might guess, this example is based heavily on the Forth language. I highly recommend the free book "Thinking Forth", which is not just about the Forth language, but about (*non* object-oriented!) problem decomposition techniques which are equally applicable to functional programming.

I got the idea for this post from a great blog by Ashley Feniello. If you want to go deeper into emulating a stack based language in F#, start there. Have fun!

In this series of posts we'll look at how functions and values are combined into expressions, and the different kinds of expressions that are available in F#.

- Expressions and syntax: Introduction. How to code in F#.
- Expressions vs. statements. Why expressions are safer and make better building blocks.
- Overview of F# expressions. Control flows, lets, dos, and more.
- Binding with let, use, and do. How to use them.
- F# syntax: indentation and verbosity. Understanding the offside rule.
- Parameter and value naming conventions. a, f, x and friends.
- Control flow expressions. And how to avoid using them.
- Exceptions. Syntax for throwing and catching.
- Match expressions. The workhorse of F#.
- Formatted text using printf. Tips and techniques for printing and logging.
- Worked example: Parsing command line arguments. Pattern matching in practice.
- Worked example: Roman numerals. More pattern matching in practice.

Expressions and syntax: Introduction

NOTE: Before reading this series, I suggest that you read the "thinking functionally" series as a prerequisite.

In this series we'll look at how functions and values are combined into expressions, and the different kinds of expressions that are available in F#.

We'll also look at some other basic topics, such as `let` bindings, F# syntax, pattern matching, and outputting text with `printf`.

This series is not meant to be exhaustive or definitive. Much of the syntax and usage of F# should be obvious from the examples, and the MSDN documentation has all the details if you need them. Rather we will just focus on explaining some of the essential areas that might be confusing.

So, we'll start with some general tips, talk about how `let` bindings work, and explain the indentation rules.

After that, the next few posts will cover `match..with` expressions, the imperative control flow expressions, and exception expressions. Computation expressions and object-oriented expressions will be left to later series.

Finally, we'll finish with some worked examples that use pattern matching as an integral part of their design.

Expressions vs. statements

In programming language terminology, an "expression" is a combination of values and functions that are combined and interpreted by the compiler to create a new value, as opposed to a "statement" which is just a standalone unit of execution and doesn't return anything. One way to think of this is that the purpose of an expression is to create a value (with some possible side-effects), while the sole purpose of a statement is to have side-effects.

C# and most imperative languages make a distinction between expressions and statements and have rules about where each kind can be used. But as should be apparent, a truly pure functional language cannot support statements at all, because in a truly pure language, there would be no side-effects.

Even though F# is not pure, it does follow the same principle. In F# everything is an expression. Not just values and functions, but also control flows (such as if-then-else and loops), pattern matching, and so on.

There are some subtle benefits to using expressions over statements. First, unlike statements, smaller expressions can be combined (or "composed") into larger expressions. So if everything is an expression, then everything is also composable.

Second, a series of statements always implies a specific order of evaluation, which means that a statement cannot be understood without looking at prior statements. But with pure expressions, the subexpressions do not have any implied order of execution or dependencies.

So in the expression `a+b`, if both the '`a`' and '`b`' parts are pure, then the '`a`' part can be isolated, understood, tested and evaluated on its own, as can the '`b`' part.
This "isolatibility" of expressions is another beneficial aspect of functional programming.

> Note that the F# interactive window also relies on everything being an expression. It would be much harder to use a C# interactive window.

Expressions are safer and more compact

Using expressions consistently leads to code that is both safer and more compact. Let's see what I mean by this.

First, let's look at a statement based approach. Statements don't return values, so you have to use temporary variables that are assigned to from within statement bodies. Here are some examples using a C-like language (OK, C#) rather than F#:

```
public void IfThenElseStatement(bool aBool)
{
   int result;     //what is the value of result before it is used?
   if (aBool)
   {
      result = 42; //what is the result in the 'else' case?
   }
   Console.WriteLine("result={0}", result);
}
```

Because the "if-then" is a statement, the `result` variable must be defined *outside* the statement and but assigned to *inside* the statement, which leads to some issues:

- The `result` variable has to be set up outside the statement itself. What initial value should it be set to?
- What if I forget to assign to the `result` variable in the `if` statement? The purpose of the "if "statement is purely to have side effects (the assignment to the variables). This means that the statements are potentially buggy, because it would be easy to forget to do an assignment in one branch. And because the assignment was just a side effect, the compiler could not offer any warning. Since the `result` variable has already been defined in scope, I could easily use it, unaware that it was invalid.
- What is the value of the `result` variable in the "else" case? In this case, I haven't specified a value. Did I forget? Is this a potential bug?
- Finally, the reliance on side-effects to get things done means that the statements are not easily usable in another context (for example, extracted for refactoring, or parallelizing) because they have a dependency on a variable that is not part of the statement itself.

Note: the code above will not compile in C# because the compiler will complain if you use an unassigned local variable like this. But having to define *some* default value for `result` before it is even used is still a problem.

For comparison, here is the same code, rewritten in an expression-oriented style:

```
public void IfThenElseExpression(bool aBool)
{
    int result = aBool ? 42 : 0;
    Console.WriteLine("result={0}", result);
}
```

In the expression-oriented version, none of the earlier issues even apply!

- The `result` variable is declared at the same time that it is assigned. No variables have to be set up "outside" the expression and there is no worry about what initial value they should be set to.
- The "else" is explicitly handled. There is no chance of forgetting to do an assignment in one of the branches.
- And I cannot possibly forget to assign to `result`, because then the variable would not even exist!

In F#, the two examples would be written as:

```
let IfThenElseStatement aBool =
   let mutable result = 0       // mutable keyword required
   if (aBool) then result <- 42
   printfn "result=%i" result
```

The " `mutable` " keyword is considered a code smell in F#, and is discouraged except in certain special cases. It should be avoided at all cost while you are learning!

In the expression based version, the mutable variable has been eliminated and there is no reassignment anywhere.

```
let IfThenElseExpression aBool =
   let result = if aBool then 42 else 0
                // note that the else case must be specified
   printfn "result=%i" result
```

Once we have the `if` statement converted into an expression, it is now trivial to refactor it and move the entire subexpression to a different context without introducing errors.

Here's the refactored version in C#:

```
public int StandaloneSubexpression(bool aBool)
{
    return aBool ? 42 : 0;
}

public void IfThenElseExpressionRefactored(bool aBool)
{
    int result = StandaloneSubexpression(aBool);
    Console.WriteLine("result={0}", result);
}
```

And in F#:

```
let StandaloneSubexpression aBool =
   if aBool then 42 else 0

let IfThenElseExpressionRefactored aBool =
   let result = StandaloneSubexpression aBool
   printfn "result=%i" result
```

Statements vs. expressions for loops

Going back to C# again, here is a similar example of statements vs. expressions using a loop statement

```
public void LoopStatement()
{
    int i;    //what is the value of i before it is used?
    int length;
    var array = new int[] { 1, 2, 3 };
    int sum;  //what is the value of sum if the array is empty?

    length = array.Length;   //what if I forget to assign to length?
    for (i = 0; i < length; i++)
    {
        sum += array[i];
    }

    Console.WriteLine("sum={0}", sum);
}
```

I've used an old-style "for" statement, where the index variables are declared outside the loop. Many of the issues discussed earlier apply to the loop index " `i` " and the max value " `length` ", such as: can they be used outside the loop? And what happens if they are not assigned to?

A more modern version of a for-loop addresses these issues by declaring and assigning the loop variables in the "for" loop itself, and by requiring the " `sum` " variable to be initialized:

```
public void LoopStatementBetter()
{
    var array = new int[] { 1, 2, 3 };
    int sum = 0;        // initialization is required

    for (var i = 0; i < array.Length; i++)
    {
        sum += array[i];
    }

    Console.WriteLine("sum={0}", sum);
}
```

This more modern version follows the general principle of combining the declaration of a local variable with its first assignment.

But of course, we can keep improving by using a `foreach` loop instead of a `for` loop:

```
public void LoopStatementForEach()
{
    var array = new int[] { 1, 2, 3 };
    int sum = 0;        // initialization is required

    foreach (var i in array)
    {
        sum += i;
    }

    Console.WriteLine("sum={0}", sum);
}
```

Each time, not only are we condensing the code, but we are reducing the likelihood of errors.

But taking that principle to its logical conclusion leads to a completely expression based approach! Here's how it might be done using LINQ:

```
public void LoopExpression()
{
    var array = new int[] { 1, 2, 3 };

    var sum = array.Aggregate(0, (sumSoFar, i) => sumSoFar + i);

    Console.WriteLine("sum={0}", sum);
}
```

Note that I could have used LINQ's built-in "sum" function, but I used `Aggregate` in order to show how the sum logic embedded in a statement can be converted into a lambda and used as part of an expression.

In the next post, we'll look at the various kinds of expressions in F#.

Overview of F# expressions

In this post we'll look at the different kinds of expressions that are available in F# and some general tips for using them.

Is everything really an expression?

You might be wondering how "everything is an expression" actually works in practice.

Let's start with some basic expression examples that should be familiar:

```
1                             // literal
[1;2;3]                       // list expression
-2                            // prefix operator
2 + 2                         // infix operator
"string".Length               // dot lookup
printf "hello"                // function application
```

No problems there. Those are obviously expressions.

But here are some more complex things which are *also* expressions. That is, each of these returns a value that can be used for something else.

```
fun () -> 1                   // lambda expression

match 1 with                  // match expression
    | 1 -> "a"
    | _ -> "b"

if true then "a" else "b"     // if-then-else

for i in [1..10]              // for loop
  do printf "%i" i

try                           // exception handling
  let result = 1 / 0
  printfn "%i" result
with
  | e ->
     printfn "%s" e.Message


let n=1 in n+2                // let expression
```

In other languages, these might be statements, but in F# they really do return values, as you can see by binding a value to the result:

```
let x1 = fun () -> 1

let x2 = match 1 with
         | 1 -> "a"
         | _ -> "b"

let x3 = if true then "a" else "b"

let x4 = for i in [1..10]
          do printf "%i" i

let x5 = try
            let result = 1 / 0
            printfn "%i" result
         with
            | e ->
                printfn "%s" e.Message


let x6 = let n=1 in n+2
```

What kinds of expressions are there?

There are lots of diffent kinds of expressions in F#, about 50 currently. Most of them are trivial and obvious, such as literals, operators, function application, "dotting into", and so on.

The more interesting and high-level ones can be grouped as follows:

- Lambda expressions
- "Control flow" expressions, including:
  - The match expression (with the `match..with` syntax)
  - Expressions related to imperative control flow, such as if-then-else, loops
  - Exception-related expressions
- "let" and "use" expressions
- Computation expressions such as `async {..}`
- Expressions related to object-oriented code, including casts, interfaces, etc

We have already discussed lambdas in the "thinking functionally" series, and as noted earlier, computation expressions and object-oriented expressions will be left to later series.

So, in upcoming posts in this series, we will focus on "control flow" expressions and "let" expressions.

"Control flow" expressions

In imperative languages, control flow expressions like if-then-else, for-in-do, and match-with are normally implemented as statements with side-effects, In F#, they are all implemented as just another type of expression.

In fact, it is not even helpful to think of "control flow" in a functional language; the concept doesn't really exist. Better to just think of the program as a giant expression containing sub-expressions, some of which are evaluated and some of which are not. If you can get your head around this way of thinking, you have a good start on thinking functionally.

There will be some upcoming posts on these different types of control flow expressions:

- The match expression
- Imperative control flow: if-then-else and for loops
- Exceptions

"let" bindings as expressions

What about `let x=something` ? In the examples above we saw:

```
let x5 = let n=1 in n+2
```

How can " `let` " be an expression? The reason will be discussed in the next post on "let", "use" and "do".

General tips for using expressions

But before we cover the important expression types in details, here are some tips for using expressions in general.

Multiple expressions on one line

Normally, each expression is put on a new line. But you can use a semicolon to separate expressions on one line if you need to. Along with its use as a separator for list and record elements, this is one of the few times where a semicolon is used in F#.

```
let f x =                           // one expression per line
      printfn "x=%i" x
      x + 1

let f x = printfn "x=%i" x; x + 1   // all on same line with ";"
```

The rule about requiring unit values until the last expression still applies, of course:

```
let x = 1;2              // error: "1;" should be a unit expression
let x = ignore 1;2       // ok
let x = printf "hello";2 // ok
```

Understanding expression evaluation order

In F#, expressions are evaluated from the "inside out" -- that is, as soon as a complete subexpression is "seen", it is evaluated.

Have a look at the following code and try to guess what will happen, then evaluate the code and see.

```
// create a clone of if-then-else
let test b t f = if b then t else f

// call it with two different choices
test true (printfn "true") (printfn "false")
```

What happens is that both "true" and "false" are printed, even though the test function will never actually evaluate the "else" branch. Why? Because the `(printfn "false")` expression is evaluated immediately, regardless of how the test function will be using it.

This style of evaluation is called "eager". It has the advantage that it is easy to understand, but it does mean that it can be inefficient on occasion.

The alternative style of evaluation is called "lazy", whereby expressions are only evaluated when they are needed. The Haskell language follows this approach, so a similar example in Haskell would only print "true".

In F#, there are a number of techniques to force expressions *not* to be evaluated immediately. The simplest it to wrap it in a function that only gets evaluated on demand:

```
// create a clone of if-then-else that accepts functions rather than simple values
let test b t f = if b then t() else f()

// call it with two different functions
test true (fun () -> printfn "true") (fun () -> printfn "false")
```

The problem with this is that now the "true" function might be evaluated twice by mistake, when we only wanted to evaluate it once!

So, the preferred way for expressions not to be evaluated immediately is to use the `Lazy<>` wrapper.

```
// create a clone of if-then-else with no restrictions...
let test b t f = if b then t else f

// ...but call it with lazy values
let f = test true (lazy (printfn "true")) (lazy (printfn "false"))
```

The final result value `f` is also a lazy value, and can be passed around without being evaluated until you are finally ready to get the result.

```
f.Force()     // use Force() to force the evaluation of a lazy value
```

If you never need the result, and never call `Force()` , then the wrapped value will never be evaluated.

There will much more on laziness in an upcoming series on performance.

Binding with let, use, and do

As we've have already seen, there are no "variables" in F#. Instead there are values.

And we have also seen that keywords such as `let` , `use` , and `do` act as *bindings* -- associating an identifier with a value or function expression.

In this post we'll look at these bindings in more detail.

"let" bindings

The `let` binding is straightforward, it has the general form:

```
let aName = someExpression
```

But there are two uses of `let` that are subtly different. One is to define a named expression at a the top level of a module*, and the other is to define a local name used in the context of some expression. This is somewhat analogous to the difference between "top level" method names and "local" variable names in C#.

* and in a later series, when we talk about OO features, classes can have top level let bindings too.

Here's an example of both types:

```
module MyModule =

    let topLevelName =
        let nestedName1 = someExpression
        let nestedName2 = someOtherExpression
        finalExpression
```

The top level name is a *definition*, which is part of the module, and you can access it with a fully qualified name such as `MyModule.topLevelName` . It's the equivalent of a class method, in a sense.

But the nested names are completely inaccessible to anyone -- they are only valid within the context of the top level name binding.

Patterns in "let" bindings

We have already seen examples of how bindings can use patterns directly

```
let a,b = 1,2

type Person = {First:string; Last:string}
let alice = {First="Alice"; Last="Doe"}
let {First=first} = alice
```

And in function definitions the binding includes parameters as well:

```
// pattern match the parameters
let add (x,y) = x + y

// test
let aTuple = (1,2)
add aTuple
```

The details of the various pattern bindings depends on the type being bound, and will be discussed further in later posts on pattern matching.

Nested "let" bindings as expressions

We have emphasized that an expression is composed from smaller expressions. But what about a nested `let` ?

```
let nestedName = someExpression
```

How can " `let` " be an expression? What does it return?

The answer that a nested "let" can never be used in isolation -- it must always be part of a larger code block, so that it can be interpreted as:

```
let nestedName = [some expression] in [some other expression involving nestedName]
```

That is, every time you see the symbol "nestedName" in the second expression (called the *body expression*), substitute it with the first expression.

So for example, the expression:

```
// standard syntax
let f () =
  let x = 1
  let y = 2
  x + y          // the result
```

really means:

```
// syntax using "in" keyword
let f () =
  let x = 1 in   // the "in" keyword is available in F#
    let y = 2 in
      x + y      // the result
```

When the substitutions are performed, the last line becomes:

```
(definition of x) + (definition of y)
// or
(1) + (2)
```

In a sense, the nested names are just "macros" or "placeholders" that disappear when the expression is compiled. And therefore you should be able to see that the nested `let`s have no effect on the expression as whole. So, for example, the type of an expression containing nested `let`s is just the type of the final body expression.

If you understand how nested `let` bindings work, then certain errors become understandable. For example, if there is nothing for a nested "let" to be "in", the entire expression is not complete. In the example below, there is nothing following the let line, which is an error:

```
let f () =
  let x = 1
// error FS0588: Block following this 'let' is unfinished.
//               Expect an expression.
```

And you cannot have multiple expression results, because you cannot have multiple body expressions. Anything evaluated before the final body expression must be a " `do` " expression (see below), and return `unit` .

```
let f () =
  2 + 2       // warning FS0020: This expression should
              // have type 'unit'
  let x = 1
  x + 1       // this is the final result
```

In a case like this, you must pipe the results into "ignore".

```
let f () =
  2 + 2 |> ignore
  let x = 1
  x + 1       // this is the final result
```

"use" bindings

The `use` keyword serves the same purpose as `let` -- it binds the result of an expression to a named value.

The key difference is that is also *automatically disposes* the value when it goes out of scope.

Obviously, this means that `use` only applies in nested situations. You cannot have a top level `use` and the compiler will warn you if you try.

```
module A =
    use f () =  // Error
      let x = 1
      x + 1
```

To see how a proper `use` binding works, first let's create a helper function that creates an `IDisposable` on the fly.

```
// create a new object that implements IDisposable
let makeResource name =
   { new System.IDisposable
     with member this.Dispose() = printfn "%s disposed" name }
```

Now let's test it with a nested `use` binding:

```
let exampleUseBinding name =
    use myResource = makeResource name
    printfn "done"

//test
exampleUseBinding "hello"
```

We can see that "done" is printed, and then immediately after that, `myResource` goes out of scope, its `Dispose` is called, and "hello disposed" is also printed.

On the other hand, if we test it using the regular `let` binding, we don't get the same effect.

```
let exampleLetBinding name =
    let myResource = makeResource name
    printfn "done"

//test
exampleLetBinding "hello"
```

In this case, we see that "done" is printed, but `Dispose` is never called.

"Use" only works with IDisposables

Note that "use" bindings only work with types that implement `IDisposable`, and the compiler will complain otherwise:

```
let exampleUseBinding2 name =
    use s = "hello"  // Error: The type 'string' is
                     // not compatible with the type 'IDisposable'
    printfn "done"
```

Don't return "use'd" values

It is important to realize that the value is disposed as soon as it goes out of scope *in the expression where it was declared*. If you attempt to return the value for use by another function, the return value will be invalid.

The following example shows how *not* to do it:

```
let returnInvalidResource name =
    use myResource = makeResource name
    myResource // don't do this!

// test
let resource = returnInvalidResource  "hello"
```

If you need to work with a disposable "outside" the function that created it, probably the best way is to use a callback.

The function then would work as follows:

- create the disposable.
- evaluate the callback with the disposable
- call `Dispose` on the disposable

Here's an example:

```
let usingResource name callback =
    use myResource = makeResource name
    callback myResource
    printfn "done"

let callback aResource = printfn "Resource is %A" aResource
do usingResource "hello" callback
```

This approach guarantees that the same function that creates the disposable also disposes of it and there is no chance of a leak.

Another possible way is to *not* use a `use` binding on creation, but use a `let` binding instead, and make the caller responsible for disposing.

Here's an example:

```
let returnValidResource name =
    // "let" binding here instead of "use"
    let myResource = makeResource name
    myResource // still valid

let testValidResource =
    // "use" binding here instead of "let"
    use resource = returnValidResource  "hello"
    printfn "done"
```

Personally, I don't like this approach, because it is not symmetrical and separates the create from the dispose, which could lead to resource leaks.

The "using" function

The preferred approach to sharing a disposable, shown above, used a callback function.

There is a built-in `using` function that works in the same way. It takes two parameters:

- the first is an expression that creates the resource
- the second is a function that uses the resource, taking it as a parameter

Here's our earlier example rewritten with the `using` function:

```
let callback aResource = printfn "Resource is %A" aResource
using (makeResource "hello") callback
```

In practice, the `using` function is not used that often, because it is so easy to make your own custom version of it, as we saw earlier.

Misusing "use"

One trick in F# is to appropriate the `use` keyword to do any kind of "stop" or "revert" functionality automatically.

The way to do this is:

- Create an extension method for some type
- In that method, start the behavior you want but then return an `IDisposable` that stops the behavior.

For example, here is an extension method that starts a timer and then returns an `IDisposable` that stops it.

```
module TimerExtensions =

    type System.Timers.Timer with
        static member StartWithDisposable interval handler =
            // create the timer
            let timer = new System.Timers.Timer(interval)

            // add the handler and start it
            do timer.Elapsed.Add handler
            timer.Start()

            // return an IDisposable that calls "Stop"
            { new System.IDisposable with
                member disp.Dispose() =
                    do timer.Stop()
                    do printfn "Timer stopped"
                }
```

So now in the calling code, we create the timer and bind it with `use`. When the timer value goes out of scope, it will stop automatically!

```
open TimerExtensions
let testTimerWithDisposable =
    let handler = (fun _ -> printfn "elapsed")
    use timer = System.Timers.Timer.StartWithDisposable 100.0 handler
    System.Threading.Thread.Sleep 500
```

This same approach can be used for other common pairs of operations, such as:

- opening/connecting and then closing/disconnecting a resource (which is what `IDisposable` is supposed to be used for anyway, but your target type might not have implemented it)
- registering and then deregistering an event handler (instead of using `WeakReference`)
- in a UI, showing a splash screen at the start of a block of code, and then automatically closing it at the end of the block

I wouldn't recommend this approach generally, because it does hide what is going on, but on occasion it can be quite useful.

"do" bindings

Sometimes we might want to execute code independently of a function or value definition. This can be useful in module initialization, class initialization and so on.

That is, rather than having "`let x = do something`" we just the "`do something`" on its own. This is analogous to a statement in an imperative language.

You can do this by prefixing the code with "`do`":

```
do printf "logging"
```

In many situations, the `do` keyword can be omitted:

```
printf "logging"
```

But in both cases, the expression must return unit. If it does not, you will get a compiler error.

```
do 1 + 1    // warning: This expression is a function
```

As always, you can force a non-unit result to be discarded by piping the results into "`ignore`".

```
do ( 1+1 |> ignore )
```

You will also see the "`do`" keyword used in loops in the same way.

Note that although you can sometimes omit it, it is considered good practice to always have an explicit "`do`", as it acts as documentation that you do not want a result, only the side-effects.

"do" for module initialization

Just like `let`, `do` can be used both in a nested context, and at the top level in a module or class.

When used at the module level, the `do` expression is evaluated once only, when the module is first loaded.

```
module A =

    module B =
        do printfn "Module B initialized"

    module C =
        do printfn "Module C initialized"

    do printfn "Module A initialized"
```

This is somewhat analogous to a static class constructor in C#, except that if there are multiple modules, the order of initialization is fixed and they are initialized in order of declaration.

let! and use! and do!

When you see `let!`, `use!` and `do!` (that is, with exclamation marks) and they are part of a curly brace `{..}` block, then they are being used as part of a "computation expression". The exact meaning of `let!`, `use!` and `do!` in this context depends on the computation expression itself. Understanding computation expressions in general will have to wait for a later series.

The most common type of computation expression you will run into are *asynchronous workflows*, indicated by a `async{..}` block. In this context, it means they are being used to wait for an async operation to finish, and only then bind to the result value.

Here are some examples we saw earlier in a post from the "why use F#?" series:

```
//This simple workflow just sleeps for 2 seconds.
open System
let sleepWorkflow  = async{
    printfn "Starting sleep workflow at %O" DateTime.Now.TimeOfDay

    // do! means to wait as well
    do! Async.Sleep 2000
    printfn "Finished sleep workflow at %O" DateTime.Now.TimeOfDay
    }

//test
Async.RunSynchronously sleepWorkflow


// Workflows with other async workflows nested inside them.
/// Within the braces, the nested workflows can be blocked on by using the let! or use
! syntax.
let nestedWorkflow  = async{

    printfn "Starting parent"

    // let! means wait and then bind to the childWorkflow value
    let! childWorkflow = Async.StartChild sleepWorkflow

    // give the child a chance and then keep working
    do! Async.Sleep 100
    printfn "Doing something useful while waiting "

    // block on the child
    let! result = childWorkflow

    // done
    printfn "Finished parent"
    }

// run the whole workflow
Async.RunSynchronously nestedWorkflow
```

Attributes on let and do bindings

If they are at the top-level in a module, `let` and `do` bindings can have attributes. F# attributes use the syntax `[<MyAttribute>]`.

Here are some examples in C# and then the same code in F#:

```
class AttributeTest
{
    [Obsolete]
    public static int MyObsoleteFunction(int x, int y)
    {
        return x + y;
    }

    [CLSCompliant(false)]
    public static void NonCompliant()
    {
    }
}
```

```
module AttributeTest =
    [<Obsolete>]
    let myObsoleteFunction x y = x + y

    [<CLSCompliant(false)>]
    let nonCompliant () = ()
```

Let's have a brief look at three attribute examples:

- The EntryPoint attribute used to indicate the "main" function.
- The various AssemblyInfo attributes.
- The DllImport attribute for interacting with unmanaged code.

The EntryPoint attribute

The special `EntryPoint` attribute is used to mark the entry point of a standalone app, just as in C#, the `static void Main` method is.

Here's the familiar C# version:

```
class Program
{
    static int Main(string[] args)
    {
        foreach (var arg in args)
        {
            Console.WriteLine(arg);
        }

        //same as Environment.Exit(code)
        return 0;
    }
}
```

And here's the F# equivalent:

```
module Program

[<EntryPoint>]
let main args =
    args |> Array.iter printfn "%A"

    0  // return is required!
```

Just as in C#, the args are an array of strings. But unlike C#, where the static `Main` method can be `void`, the F# function *must* return an int.

Also, a big gotcha is that the function that has this attribute must be the very last function in the last file in the project! Otherwise you get this error:

```
error FS0191: A function labelled with the 'EntryPointAttribute' atribute must be the
last declaration in the last file in the compilation sequence
```

Why is the F# compiler so fussy? In C#, the class can go anywhere.

One analogy that might help is this: in some sense, the whole application is a single huge expression bound to `main`, where `main` is an expression that contains subexpressions that contain other subexpressions.

```
[<EntryPoint>]
let main args =
    the entire application as a set of subexpressions
```

Now in F# projects, there are no forward references allowed. That is, expressions that refer to other expressions must be declared after them. And so logically, the highest, most top-level function of them all, `main`, must come last of all.

The AssemblyInfo attributes

In a C# project, there is an `AssemblyInfo.cs` file that contains all the assembly level attributes.

In F#, the equivalent way to do this is with a dummy module which contains a `do` expression annotated with these attributes.

```
open System.Reflection

module AssemblyInfo =
    [<assembly: AssemblyTitle("MyAssembly")>]
    [<assembly: AssemblyVersion("1.2.0.0")>]
    [<assembly: AssemblyFileVersion("1.2.3.4152")>]
    do ()   // do nothing -- just a placeholder for the attribute
```

The DllImport attribute

Another occasionally useful attribute is the `DllImport` attribute. Here's a C# example.

```
using System.Runtime.InteropServices;

[TestFixture]
public class TestDllImport
{
    [DllImport("shlwapi", CharSet = CharSet.Auto, EntryPoint = "PathCanonicalize", Set
LastError = true)]
    private static extern bool PathCanonicalize(StringBuilder lpszDst, string lpszSrc)
;

    [Test]
    public void TestPathCanonicalize()
    {
        var input = @"A:\name_1\.\name_2\..\name_3";
        var expected = @"A:\name_1\name_3";

        var builder = new StringBuilder(260);
        PathCanonicalize(builder, input);
        var actual = builder.ToString();

        Assert.AreEqual(expected,actual);
    }
}
```

It works the same way in F# as in C#. One thing to note is that the `extern declaration ...` puts the types before the parameters, C-style.

```
open System.Runtime.InteropServices
open System.Text

[<DllImport("shlwapi", CharSet = CharSet.Ansi, EntryPoint = "PathCanonicalize", SetLas
tError = true)>]
extern bool PathCanonicalize(StringBuilder lpszDst, string lpszSrc)

let TestPathCanonicalize() =
    let input = @"A:\name_1\.\name_2\..\name_3"
    let expected = @"A:\name_1\name_3"

    let builder = new StringBuilder(260)
    let success = PathCanonicalize(builder, input)
    let actual = builder.ToString()

    printfn "actual=%s success=%b" actual (expected = actual)

// test
TestPathCanonicalize()
```

Interop with unmanaged code is a big topic which will need its own series.

F# syntax: indentation and verbosity

The syntax for F# is mostly straightforward. But there are a few rules that you should understand if you want to avoid common indentation errors. If you are familiar with a language like Python that also is whitespace sensitive, be aware that that the rules for indentation in F# are subtly different.

Indentation and the "offside" rule

In soccer, the offside rule says that in some situations, a player cannot be "ahead" of the ball when they should be behind or level with it. The "offside line" is the line the player must not cross. F# uses the same term to describe the line at which indentation must start. As with soccer, the trick to avoiding a penalty is to know where the line is and not get ahead of it.

Generally, once an offside line has been set, all the expressions must align with the line.

```
//character columns
//3456789
let f =
  let x=1     // offside line is at column 3
  let y=1     // this line must start at column 3
  x+y         // this line must start at column 3

let f =
  let x=1     // offside line is at column 3
   x+1        // oops! don't start at column 4
              // error FS0010: Unexpected identifier in binding

let f =
  let x=1    // offside line is at column 3
 x+1         // offside! You are ahead of the ball!
             // error FS0588: Block following this
             // 'let' is unfinished
```

Various tokens can trigger new offside lines to be created. For example, when the F# sees the "`=`" used in a let expression, a new offside line is created at the position of the very next symbol or word encountered.

```
//character columns
//34567890123456789
let f =   let x=1  // line is now at column 11 (start of "let x=")
          x+1      // must start at column 11 from now on

//        |        // offside line at col 11
let f =   let x=1  // line is now at column 11 (start of "let x=")
         x+1       // offside!


// |       // offside line at col 4
let f =
   let x=1  // first word after = sign defines the line
            // offside line is now at column 4
   x+1      // must start at column 4 from now on
```

Other tokens have the same behavior, including parentheses, " `then` ", " `else` ", " `try` ", " `finally` " and " `do` ", and " `->` " in match clauses.

```
//character columns
//34567890123456789
let f =
   let g = (
    1+2)              // first char after "(" defines
                      // a new line at col 5
   g

let f =
   if true then
    1+2              // first char after "then" defines
                     // a new line at col 5

let f =
   match 1 with
   | 1 ->
       1+2           // first char after match "->" defines
                     // a new line at col 8
```

The offside lines can be nested, and are pushed and popped as you would expect:

```
//character columns
//34567890123456789
let f =
   let g = let x = 1 // first word after "let g ="
                     // defines a new offside line at col 12
           x + 1     // "x" must align at col 12
                     // pop the offside line stack now
   g + 1             // back to previous line. "g" must align
                     // at col 4
```

New offside lines can never go forward further than the previous line on the stack:

```
let f =
   let g = (         // let defines a new line at col 4
  1+2)               // oops! Cant define new line less than 4
   g
```

Special cases

There are number of special cases which have been created to make code formatting more flexible. Many of them will seem natural, such as aligning the start of each part of an `if-then-else` expression or a `try-catch` expression. There are some non-obvious ones, however.

Infix operators such as "+", "|>" and ">>" are allowed to be outside the line by their length plus one space:

```
//character columns
//34567890123456789
let x =  1   // defines a new line at col 10
       + 2   // "+" allowed to be outside the line
       + 3

let f g h =   g   // defines a new line at col 15
           >> h   // ">>" allowed to be outside the line
```

If an infix operator starts a line, that line does not have to be strict about the alignment:

```
let x =  1   // defines a new line at col 10
        + 2   // infix operators that start a line don't count
             * 3  // starts with "*" so doesn't need to align
         - 4  // starts with "-" so doesn't need to align
```

If a " `fun` " keyword starts an expression, the "fun" does *not* start a new offside line:

```
//character columns
//34567890123456789
let f = fun x ->  // "fun" should define a new line at col 9
   let y = 1      // but doesn't. The real line starts here.
   x + y
```

Finding out more

There are many more details as to how indentation works, but the examples above should cover most of the common cases. If you want to know more, the complete language spec for F# is available from Microsoft as a downloadable PDF, and is well worth reading.

"Verbose" syntax

By default, F# uses indentation to indicate block structure -- this is called "light" syntax. There is an alternative syntax that does not use indentation; it is called "verbose" syntax. With verbose syntax, you are not required to use indentation, and whitespace is not significant, but the downside is that you are required to use many more keywords, including things like:

- "`in`" keywords after every "let" and "do" binding
- "`begin`"/"`end`" keywords for code blocks such as if-then-else
- "`done`" keywords at the end of loops
- keywords at the beginning and end of type definitions

Here is an example of verbose syntax with wacky indentation that would not otherwise be acceptable:

```
#indent "off"

      let f =
    let x = 1 in
  if x=2 then
begin "a" end else begin
"b"
end

#indent "on"
```

Verbose syntax is always available, even in "light" mode, and is occasionally useful. For example, when you want to embed "let" into a one line expression:

```
let x = let y = 1 in let z = 2 in y + z
```

Other cases when you might want to use verbose syntax are:

- when outputting generated code
- to be compatible with OCaml
- if you are visually impaired or blind and use a screen reader
- or just to gain some insight into the abstract syntax tree used by the F# parser

Other than these cases, verbose syntax is rarely used in practice.

Parameter and value naming conventions

If you are coming to F# from an imperative language such as C#, then you might find a lot of the names shorter and more cryptic than you are used to.

In C# and Java, the best practice is to have long descriptive identifiers. In functional languages, the function names themselves can be descriptive, but the local identifiers inside a function tend to be quite short, and piping and composition is used a lot to get everything on a minimal number of lines.

For example, here is a crude implementation of a prime number sieve with very descriptive names for the local values.

```
let primesUpTo n =
    // create a recursive intermediate function
    let rec sieve listOfNumbers  =
        match listOfNumbers with
        | [] -> []
        | primeP::sievedNumbersBiggerThanP->
            let sievedNumbersNotDivisibleByP =
                sievedNumbersBiggerThanP
                |> List.filter (fun i-> i % primeP > 0)
            //recursive part
            let newPrimes = sieve sievedNumbersNotDivisibleByP
            primeP :: newPrimes
    // use the sieve
    let listOfNumbers = [2..n]
    sieve listOfNumbers     // return

//test
primesUpTo 100
```

Here is the same implementation, with terser, idiomatic names and more compact code:

```
let primesUpTo n =
   let rec sieve l  =
      match l with
      | [] -> []
      | p::xs ->
            p :: sieve [for x in xs do if (x % p) > 0 then yield x]
   [2..n] |> sieve
```

The cryptic names are not always better, of course, but if the function is kept to a few lines and the operations used are standard, then this is a fairly common idiom.

The common naming conventions are as follows:

- "a", "b", "c" etc., are types
- "f", "g", "h" etc., are functions
- "x", "y", "z" etc., are arguments to the functions
- Lists are indicated by adding an "s" suffix, so that " `xs` " is a list of `x` 's, " `fs` " is a list of functions, and so on. It is extremely common to see " `x::xs` " meaning the head (first element) and tail (the remaining elements) of a list.
- "*" is used whenever you don't care about the value. So "`x::* `" means that you don't care about the rest of the list, and "` let f _ = something `" means you don't care about the argument to `f`.

Another reason for the short names is that often, they cannot be assigned to anything meaningful. For example, the definition of the pipe operator is:

```
let (|>) x f = f x
```

We don't know what `f` and `x` are going to be, `f` could be any function and `x` could be any value. Making this explicit does not make the code any more understandable.

```
let (|>) aValue aFunction = aFunction aValue // any better?
```

The style used on this site

On this site I will use both styles. For the introductory series, when most of the concepts are new, I will use a very descriptive style, with intermediate values and long names. But in more advanced series, the style will become terser.

Control flow expressions

In this post, we'll look at the control flow expressions, namely:

- if-then-else
- for x in collection (which is the same as foreach in C#)
- for x = start to end
- while-do

These control flow expressions are no doubt very familiar to you. But they are very "imperative" rather than functional.

So I would strongly recommend that you do not use them if at all possible, especially when you are learning to think functionally. If you do use them as a crutch, you will find it much harder to break away from imperative thinking.

To help you do this, I will start each section with examples of how to avoid using them by using more idiomatic constructs instead. If you do need to use them, there are some "gotchas" that you need to be aware of.

If-then-else

How to avoid using if-then-else

The best way to avoid `if-then-else` is to use "match" instead. You can match on a boolean, which is similar to the classic then/else branches. But much, much better, is to avoid the equality test and actually match on the thing itself, as shown in the last implementation below.

```
// bad
let f x =
    if x = 1
    then "a"
    else "b"

// not much better
let f x =
    match x=1 with
    | true -> "a"
    | false -> "b"

// best
let f x =
    match x with
    | 1 -> "a"
    | _ -> "b"
```

Part of the reason why direct matching is better is that the equality test throws away useful information that you often need to retrieve again.

This is demonstrated by the next scenario, where we want to get the first element of a list in order to print it. Obviously, we must be careful not to attempt this for an empty list.

The first implementation does a test for empty and then a *second* operation to get the first element. A much better approach is to match and extract the element in one single step, as shown in the second implementation.

```
// bad
let f list =
    if List.isEmpty list
    then printfn "is empty"
    else printfn "first element is %s" (List.head list)

// much better
let f list =
    match list with
    | [] -> printfn "is empty"
    | x::_ -> printfn "first element is %s" x
```

The second implementation is not only easier to understand, it is more efficient.

If the boolean test is complicated, it can still be done with match by using extra " `when` " clauses (called "guards"). Compare the first and second implementations below to see the difference.

```
// bad
let f list =
    if List.isEmpty list
        then printfn "is empty"
        elif (List.head list) > 0
            then printfn "first element is > 0"
            else printfn "first element is <= 0"

// much better
let f list =
    match list with
    | [] -> printfn "is empty"
    | x::_ when x > 0 -> printfn "first element is > 0"
    | x::_ -> printfn "first element is <= 0"
```

Again, the second implementation is easier to understand and also more efficient.

The moral of the tale is: if you find yourself using if-then-else or matching on booleans, consider refactoring your code.

How to use if-then-else

If you do need to use if-then-else, be aware that even though the syntax looks familiar, there is a catch that you must be aware of: " `if-then-else` " is an *expression*, not a *statement*, and as with every expression in F#, it must return a value of a particular type.

Here are two examples where the return type is a string.

```
let v = if true then "a" else "b"    // value : string
let f x = if x then "a" else "b"     // function : bool->string
```

But as a consequence, both branches must return the same type! If this is not true, then the expression as a whole cannot return a consistent type and the compiler will complain.

Here is an example of different types in each branch:

```
let v = if true then "a" else 2
  // error FS0001: This expression was expected to have
  //                 type string but here has type int
```

The "else" clause is optional, but if it is absent, the "else" clause is assumed to return unit, which means that the "then" clause must also return unit. You will get a complaint from the compiler if you make this mistake.

```
let v = if true then "a"
  // error FS0001: This expression was expected to have type unit
  //                 but here has type string
```

If the "then" clause returns unit, then the compiler will be happy.

```
let v2 = if true then printfn "a"   // OK as printfn returns unit
```

Note that there is no way to return early in a branch. The return value is the entire expression. In other words, the if-then-else expression is more closely related to the C# ternary if operator (?:) than to the C# if-then-else statement.

if-then-else for one liners

One of the places where if-then-else can be genuinely useful is to create simple one-liners for passing into other functions.

```
let posNeg x = if x > 0 then "+" elif x < 0 then "-" else "0"
[-5..5] |> List.map posNeg
```

Returning functions

Don't forget that an if-then-else expression can return any value, including function values. For example:

```
let greetings =
    if (System.DateTime.Now.Hour < 12)
    then (fun name -> "good morning, " + name)
    else (fun name -> "good day, " + name)

//test
greetings "Alice"
```

Of course, both functions must have the same type, meaning that they must have the same function signature.

Loops

How to avoid using loops

The best way to avoid loops is to use the built in list and sequence functions instead. Almost anything you want to do can be done without using explicit loops. And often, as a side benefit, you can avoid mutable values as well. Here are some examples to start with, and for more details please read the upcoming series devoted to list and sequence operations.

Example: Printing something 10 times:

```
// bad
for i = 1 to 10 do
   printf "%i" i

// much better
[1..10] |> List.iter (printf "%i")
```

Example: Summing a list:

```
// bad
let sum list =
    let mutable total = 0    // uh-oh -- mutable value
    for e in list do
        total <- total + e   // update the mutable value
    total                    // return the total

// much better
let sum list = List.reduce (+) list

//test
sum [1..10]
```

Example: Generating and printing a sequence of random numbers:

```
// bad
let printRandomNumbersUntilMatched matchValue maxValue =
  let mutable continueLooping = true  // another mutable value
  let randomNumberGenerator = new System.Random()
  while continueLooping do
    // Generate a random number between 1 and maxValue.
    let rand = randomNumberGenerator.Next(maxValue)
    printf "%d " rand
    if rand = matchValue then
       printfn "\nFound a %d!" matchValue
       continueLooping <- false

// much better
let printRandomNumbersUntilMatched matchValue maxValue =
  let randomNumberGenerator = new System.Random()
  let sequenceGenerator _ = randomNumberGenerator.Next(maxValue)
  let isNotMatch = (<>) matchValue

  //create and process the sequence of rands
  Seq.initInfinite sequenceGenerator
    |> Seq.takeWhile isNotMatch
    |> Seq.iter (printf "%d ")

  // done
  printfn "\nFound a %d!" matchValue

//test
printRandomNumbersUntilMatched 10 20
```

As with if-then-else, there is a moral; if you find yourself using loops and mutables, please consider refactoring your code to avoid them.

The three types of loops

If you want to use loops, then there are three types of loop expressions to choose from, which are similar to those in C#.

- `for-in-do` . This has the form `for x in enumerable do something` . It is the same as the `foreach` loop in C#, and is the form most commonly seen in F#.
- `for-to-do` . This has the form `for x = start to finish do something` . It is the same as the standard `for (i=start; i<end; i++)` loops in C#.
- `while-do` . This has the form `while test do something` . It is the same as the `while` loop in C#. Note that there is no `do-while` equivalent in F#.

I won't go into any more detail than this, as the usage is straightforward. If you have trouble, check the MSDN documentation.

How to use loops

As with if-then-else expressions, the loop expressions look familiar, but there are some catches again.

- All looping expressions always return unit for the whole expression, so there is no way to return a value from inside a loop.
- As with all "do" bindings, the expression inside the loop must return unit as well.
- There is no equivalent of "break" and "continue" (this can generally done better using sequences anyway)

Here's an example of the unit constraint. The expression in the loop should be unit, not int, so the compiler will complain.

```
let f =
  for i in [1..10] do
    i + i  // warning: This expression should have type 'unit'

// version 2
let f =
  for i in [1..10] do
    i + i |> ignore   // fixed
```

Loops for one liners

One of the places where loops are used in practice is as list and sequence generators.

```
let myList = [for x in 0..100 do if x*x < 100 then yield x ]
```

Summary

I'll repeat what I said at the top of the post: do avoid using imperative control flow when you are learning to think functionally. And understand the exceptions that prove the rule; the one-liners whose use is acceptable.

Exceptions

Just like other .NET languages, F# supports throwing and catching exceptions. As with the control flow expressions, the syntax will feel familiar, but again there are a few catches that you should know about.

Defining your own exceptions

When raising/throwing exceptions, you can use the standard system ones such as `InvalidOperationException`, or you can define your own exception types using the simple syntax shown below, where the "content" of the exception is any F# type:

```
exception MyFSharpError1 of string
exception MyFSharpError2 of string * int
```

That's it! Defining new exception classes is a lot easier than in C#!

Throwing exceptions

There are three basic ways to throw an exception

- Using one of the built in functions, such as "invalidArg"
- Using one of the standard .NET exception classes
- Using your own custom exception types

Throwing exceptions, method 1: using one of the built in functions

There are four useful exception keywords built into F#:

- `failwith` throws a generic `System.Exception`
- `invalidArg` throws an `ArgumentException`
- `nullArg` throws a `NullArgumentException`
- `invalidOp` throws an `InvalidOperationException`

These four probably cover most of the exceptions you would regularly throw. Here is how they are used:

```
// throws a generic System.Exception
let f x =
   if x then "ok"
   else failwith "message"

// throws an ArgumentException
let f x =
   if x then "ok"
   else invalidArg "paramName" "message"

// throws a NullArgumentException
let f x =
   if x then "ok"
   else nullArg "paramName" "message"

// throws an InvalidOperationException
let f x =
   if x then "ok"
   else invalidOp "message"
```

By the way, there's a very useful variant of `failwith` called `failwithf` that includes `printf` style formatting, so that you can make custom messages easily:

```
open System
let f x =
    if x = "bad" then
        failwithf "Operation '%s' failed at time %O" x DateTime.Now
    else
        printfn "Operation '%s' succeeded at time %O" x DateTime.Now

// test
f "good"
f "bad"
```

Throwing exceptions, method 2: using one of the standard .NET exception classes

You can `raise` any .NET exception explicitly:

```
// you control the exception type
let f x =
   if x then "ok"
   else raise (new InvalidOperationException("message"))
```

Throwing exceptions, method 3: using your own F# exception types

Finally, you can use your own types, as defined earlier.

```
// using your own F# exception types
let f x =
   if x then "ok"
   else raise (MyFSharpError1 "message")
```

And that's pretty much it for throwing exceptions.

What effect does raising an exception have on the function type?

We said earlier that both branches of an if-then-else expression must return the same type. But how can raising an exception work with this constraint?

The answer is that any code that raises exceptions is ignored for the purposes of determining expression types. This means that the function signature will be based on the normal case only, not the exception case.

For example, in the code below, the exceptions are ignored, and the overall function has signature `bool->int` , as you would expect.

```
let f x =
   if x then 42
   elif true then failwith "message"
   else invalidArg "paramName" "message"
```

Question: what do you think the function signature will be if both branches raise exceptions?

```
let f x =
   if x then failwith "error in true branch"
   else failwith "error in false branch"
```

Try it and see!

Catching exceptions

Exceptions are caught using a try-catch block, as in other languages. F# calls it `try-with` instead, and testing for each type of exception uses the standard pattern matching syntax.

```
try
    failwith "fail"
with
    | Failure msg -> "caught: " + msg
    | MyFSharpError1 msg -> " MyFSharpError1: " + msg
    | :? System.InvalidOperationException as ex -> "unexpected"
```

If the exception to catch was thrown with `failwith` (e.g. a System.Exception) or a custom F# exception, you can match using the simple tag approach shown above.

On the other hand, to catch a specific .NET exception class, you have to match using the more complicated syntax:

```
:? (exception class) as ex
```

Again, as with if-then-else and the loops, the try-with block is an expression that returns a value. This means that all branches of the `try-with` expression *must* return the same type.

Consider this example:

```
let divide x y=
    try
        (x+1) / y                      // error here -- see below
    with
    | :? System.DivideByZeroException as ex ->
          printfn "%s" ex.Message
```

When we try to evaluate it, we get an error:

```
error FS0043: The type 'unit' does not match the type 'int'
```

The reason is that the " `with` " branch is of type `unit`, while the " `try` " branch is of type `int`. So the two branches are of incompatible types.

To fix this, we need to make the " `with` " branch also return type `int`. We can do this easily using the semicolon trick to chain expressions on one line.

```
let divide x y=
    try
        (x+1) / y
    with
    | :? System.DivideByZeroException as ex ->
          printfn "%s" ex.Message; 0            // added 0 here!

//test
divide 1 1
divide 1 0
```

Now that the `try-with` expression has a defined type, the whole function can be assigned a type, namely `int -> int -> int`, as expected.

As before, if any branch throws an exception, it doesn't count when types are being determined.

Rethrowing exceptions

If needed, you can call the "`reraise()`" function in a catch handler to propagate the same exception up the call chain. This is the same as the C# `throw` keyword.

```
let divide x y=
    try
        (x+1) / y
    with
    | :? System.DivideByZeroException as ex ->
          printfn "%s" ex.Message
          reraise()

//test
divide 1 1
divide 1 0
```

Try-finally

Another familiar expression is `try-finally`. As you might expect, the "finally" clause will be called no matter what.

```
let f x =
    try
        if x then "ok" else failwith "fail"
    finally
        printf "this will always be printed"
```

The return type of the try-finally expression as a whole is always the same as return type of the "try" clause on its own. The "finally" clause has no effect on the type of the expression as a whole. So in the above example, the whole expression has type `string`.

The "finally" clause must always return unit, so any non-unit values will be flagged by the compiler.

```
let f x =
    try
        if x then "ok" else failwith "fail"
    finally
        1+1  // This expression should have type 'unit
```

Combining try-with and try-finally

The try-with and the try-finally expressions are distinct and cannot be combined directly into a single expression. Instead, you will have to nest them as circumstances require.

```
let divide x y=
   try
      try
         (x+1) / y
      finally
         printf "this will always be printed"
   with
   | :? System.DivideByZeroException as ex ->
           printfn "%s" ex.Message; 0
```

Should functions throw exceptions or return error structures?

When you are designing a function, should you throw exceptions, or return structures which encode the error? This section will discuss two different approaches.

The pair of functions approach

One approach is to provide two functions: one which assumes everything works and throws an exception otherwise and a second "tryXXX" function that returns a missing value if something goes wrong.

For example, we might want to design two distinct library functions for division, one that doesn't handle exceptions and one that does:

```
// library function that doesn't handle exceptions
let divideExn x y = x / y

// library function that converts exceptions to None
let tryDivide x y =
   try
       Some (x / y)
   with
   | :? System.DivideByZeroException -> None // return missing
```

Note the use of `Some` and `None` Option types in the `tryDivide` code to signal to the client whether the value is valid.

With the first function, the client code must handle the exception explicitly.

```
// client code must handle exceptions explicitly
try
    let n = divideExn 1 0
    printfn "result is %i" n
with
| :? System.DivideByZeroException as ex -> printfn "divide by zero"
```

Note that there is no constraint that forces the client to do this, so this approach can be a source of errors.

With the second function the client code is simpler, and the client is constrained to handle both the normal case and the error case.

```
// client code must test both cases
match tryDivide 1 0 with
| Some n -> printfn "result is %i" n
| None -> printfn "divide by zero"
```

This "normal vs. try" approach is very common in the .NET BCL, and also occurs in a few cases in the F# libraries too. For example, in the `List` module:

- `List.find` will throw a `KeyNotFoundException` if the key is not found
- But `List.tryFind` will return an Option type, with `None` if the key is not found

If you are going to use this approach, do have a naming convention. For example:

- "doSomethingExn" for functions that expect clients to catch exceptions.
- "tryDoSomething" for functions that handle normal exceptions for you.

Note that I prefer to have an "Exn" suffix on "doSomething" rather than no suffix at all. It makes it clear that you expect clients to catch exceptions even in normal cases.

The overall problem with this approach is that you have to do extra work to create pairs of functions, and you reduce the safety of the system by relying on the client to catch exceptions if they use the unsafe version of the function.

The error-code-based approach

> "Writing good error-code-based code is hard, but writing good exception-based code is really hard." *Raymond Chen*

In the functional world, returning error codes (or rather error *types*) is generally preferred to throwing exceptions, and so a standard hybrid approach is to encode the common cases (the ones that you would expect a user to care about) into a error type, but leave the very unusual exceptions alone.

Often, the simplest approach is just to use the option type: `Some` for success and `None` for errors. If the error case is obvious, as in `tryDivide` or `tryParse`, there is no need to be explicit with more detailed error cases.

But sometimes there is more than one possible error, and each should be handled differently. In this case, a union type with a case for each error is useful.

In the following example, we want to execute a SqlCommand. Three very common error cases are login errors, constraint errors and foreign key errors, so we build them into the result structure. All other errors are raised as exceptions.

```
open System.Data.SqlClient

type NonQueryResult =
    | Success of int
    | LoginError of SqlException
    | ConstraintError of SqlException
    | ForeignKeyError of SqlException

let executeNonQuery (sqlCommmand:SqlCommand) =
    try
       use sqlConnection = new SqlConnection("myconnection")
       sqlCommmand.Connection <- sqlConnection
       let result = sqlCommmand.ExecuteNonQuery()
       Success result
    with
    | :?SqlException as ex ->     // if a SqlException
        match ex.Number with
        | 18456 ->                // login Failed
            LoginError ex
        | 2601 | 2627 ->          // handle constraint error
            ConstraintError ex
        | 547 ->                  // handle FK error
            ForeignKeyError ex
        | _ ->                    // don't handle any other cases
            reraise()
       // all non SqlExceptions are thrown normally
```

The client is then forced to handle the common cases, while uncommon exceptions will be caught by a handler higher up the call chain.

```
let myCmd = new SqlCommand("DELETE Product WHERE ProductId=1")
let result =  executeNonQuery myCmd
match result with
| Success n -> printfn "success"
| LoginError ex -> printfn "LoginError: %s" ex.Message
| ConstraintError ex -> printfn "ConstraintError: %s" ex.Message
| ForeignKeyError ex -> printfn "ForeignKeyError: %s" ex.Message
```

Unlike a traditional error code approach, the caller of the function does not have to handle any errors immediately, and can simply pass the structure around until it gets to someone who knows how to handle it, as shown below:

```
let lowLevelFunction commandString =
  let myCmd = new SqlCommand(commandString)
  executeNonQuery myCmd          //returns result

let deleteProduct id =
  let commandString = sprintf "DELETE Product WHERE ProductId=%i" id
  lowLevelFunction commandString  //returns without handling errors

let presentationLayerFunction =
  let result = deleteProduct 1
  match result with
  | Success n -> printfn "success"
  | errorCase -> printfn "error %A" errorCase
```

On the other hand, unlike C#, the result of a expression cannot be accidentally thrown away. So if a function returns an error result, the caller must handle it (unless it really wants to be badly behaved and send it to `ignore`)

```
let presentationLayerFunction =
  do deleteProduct 1    // error: throwing away a result code!
```

Match expressions

Pattern matching is ubiquitous in F#. It is used for binding values to expressions with `let`, and in function parameters, and for branching using the `match..with` syntax.

We have briefly covered binding values to expressions in a post in the "why use F#?" series, and it will be covered many times as we investigate types.

So in this post, we'll cover the `match..with` syntax and its use for control flow.

What is a match expression?

We have already seen `match..with` expressions a number of times. And we know that it has the form:

```
match [something] with
| pattern1 -> expression1
| pattern2 -> expression2
| pattern3 -> expression3
```

If you squint at it just right, it looks a bit like a series of lambda expressions:

```
match [something] with
| lambda-expression-1
| lambda-expression-2
| lambda-expression-3
```

Where each lambda expression has exactly one parameter:

```
param -> expression
```

So one way of thinking about `match..with` is that it is a choice between a set of lambda expressions. But how to make the choice?

This is where the patterns come in. The choice is made based on whether the "match with" value can be matched with the parameter of the lambda expression. The first lambda whose parameter can be made to match the input value "wins"!

So for example, if the param is the wildcard `_`, it will always match, and if first, always win.

```
_ -> expression
```

Order is important!

Looking at the following example:

```
let x =
    match 1 with
    | 1 -> "a"
    | 2 -> "b"
    | _ -> "z"
```

We can see that there are three lambda expressions to match, in this order:

```
fun 1 -> "a"
fun 2 -> "b"
fun _ -> "z"
```

So, the `1` pattern gets tried first, then then the `2` pattern, and finally, the `_` pattern.

On the other hand, if we changed the order to put the wildcard first, it would be tried first and always win immediately:

```
let x =
    match 1 with
    | _ -> "z"
    | 1 -> "a"
    | 2 -> "b"
```

In this case, the F# compiler helpfully warns us that the other rules will never be matched.

So this is one major difference between a " `switch` " or " `case` " statement compared with a `match..with` . In a `match..with` , **the order is important**.

Formatting a match expression

Since F# is sensitive to indentation, you might be wondering how best to format this expression, as there are quite a few moving parts.

The post on F# syntax gives an overview of how alignment works, but for `match..with` expressions, here are some specific guidelines.

Guideline 1: The alignment of the `| expression` clauses should be directly under the `match`

This guideline is straightforward.

```
let f x =   match x with
            // aligned
            | 1 -> "pattern 1"
            // aligned
            | 2 -> "pattern 2"
            // aligned
            | _ -> "anything"
```

Guideline 2: The `match..with` should be on a new line

The `match..with` can be on the same line or a new line, but using a new line keeps the indenting consistent, independent of the lengths of the names:

```
                                              // ugly alignment!
let myVeryLongNameForAFunction myParameter =  match myParameter with
                                              | 1 -> "something"
                                              | _ -> "anything"


// much better
let myVeryLongNameForAFunction myParameter =
    match myParameter with
    | 1 -> "something"
    | _ -> "anything"
```

Guideline 3: The expression after the arrow `->` should be on a new line

Again, the result expression can be on the same line as the arrow, but using a new line again keeps the indenting consistent and helps to separate the match pattern from the result expression.

```
let f x =
    match x with
    | "a very long pattern that breaks up the flow" -> "something"
    | _ -> "anything"

let f x =
    match x with
    | "a very long pattern that breaks up the flow" ->
        "something"
    | _ ->
        "anything"
```

Of course, when all the patterns are very compact, a common sense exception can be made:

```
let f list =
    match list with
    | [] -> "something"
    | x::xs -> "something else"
```

match..with is an expression

It is important to realize that `match..with` is not really a "control flow" construct. The "control" does not "flow" down the branches, but instead, the whole thing is an expression that gets evaluated at some point, just like any other expression. The end result in practice might be the same, but it is a conceptual difference that can be important.

One consequence of it being an expression is that all branches *must* evaluate to the *same* type -- we have already seen this same behavior with if-then-else expressions and for loops.

```
let x =
    match 1 with
    | 1 -> 42
    | 2 -> true  // error wrong type
    | _ -> "hello" // error wrong type
```

You cannot mix and match the types in the expression.

You can use match expressions anywhere

Since they are normal expressions, match expressions can appear anywhere an expression can be used.

For example, here's a nested match expression:

```
// nested match..withs are ok
let f aValue =
    match aValue with
    | x ->
        match x with
        | _ -> "something"
```

And here's a match expression embedded in a lambda:

```
[2..10]
|> List.map (fun i ->
        match i with
        | 2 | 3 | 5 | 7 -> sprintf "%i is prime" i
        | _ -> sprintf "%i is not prime" i
        )
```

Exhaustive matching

Another consequence of being an expression is that there must always be *some* branch that matches. The expression as a whole must evaluate to *something*!

That is, the valuable concept of "exhaustive matching" comes from the "everything-is-an-expression" nature of F#. In a statement oriented language, there would be no requirement for this to happen.

Here's an example of an incomplete match:

```
let x =
    match 42 with
    | 1 -> "a"
    | 2 -> "b"
```

The compiler will warn you if it thinks there is a missing branch. And if you deliberately ignore the warning, then you will get a nasty runtime error (`MatchFailureException`) when none of the patterns match.

Exhaustive matching is not perfect

The algorithm for checking that all possible matches are listed is good but not always perfect. Occasionally it will complain that you have not matched every possible case, when you know that you have. In this case, you may need to add an extra case just to keep the compiler happy.

Using (and avoiding) the wildcard match

One way to guarantee that you always match all cases is to put the wildcard parameter as the last match:

```
let x =
    match 42 with
    | 1 -> "a"
    | 2 -> "b"
    | _ -> "z"
```

You see this pattern frequently, and I have used it a lot in these examples. It's the equivalent of having a catch-all `default` in a switch statement.

But if you want to get the full benefits of exhaustive pattern matching, I would encourage you *not* to use wildcards, and try to match all the cases explicitly if you can. This is particularly true if you are matching on the cases of a union type:

```
type Choices = A | B | C
let x =
    match A with
    | A -> "a"
    | B -> "b"
    | C -> "c"
    //NO default match
```

By being always explicit in this way, you can trap any error caused by adding a new case to the union. If you had a wildcard match, you would never know.

If you can't have *every* case be explicit, you might try to document your boundary conditions as much as possible, and assert an runtime error for the wildcard case.

```
let x =
    match -1 with
    | 1 -> "a"
    | 2 -> "b"
    | i when i >= 0 && i<=100 -> "ok"
    // the last case will always match
    | x -> failwithf "%i is out of range" x
```

Types of patterns

There are lots of different ways of matching patterns, which we'll look at next.

For more details on the various patterns, see the MSDN documentation.

Binding to values

The most basic pattern is to bind to a value as part of the match:

```
let y =
    match (1,0) with
    // binding to a named value
    | (1,x) -> printfn "x=%A" x
```

By the way, I have deliberately left this pattern (and others in this post) as incomplete. As an exercise, make them complete without using the wildcard.

It is important to note that the values that are bound *must* be distinct for each pattern. So you can't do something like this:

```
let elementsAreEqual aTuple =
    match aTuple with
    | (x,x) ->
        printfn "both parts are the same"
    | (_,_) ->
        printfn "both parts are different"
```

Instead, you have to do something like this:

```
let elementsAreEqual aTuple =
    match aTuple with
    | (x,y) ->
        if (x=y) then printfn "both parts are the same"
        else printfn "both parts are different"
```

This second option can also be rewritten using "guards" (`when` clauses) instead. Guards will be discussed shortly.

AND and OR

You can combine multiple patterns on one line, with OR logic and AND logic:

```
let y =
    match (1,0) with
    // OR  -- same as multiple cases on one line
    | (2,x) | (3,x) | (4,x) -> printfn "x=%A" x

    // AND  -- must match both patterns at once
    // Note only a single "&" is used
    | (2,x) & (_,1) -> printfn "x=%A" x
```

The OR logic is particularly common when matching a large number of union cases:

```
type Choices = A | B | C | D
let x =
    match A with
    | A | B | C -> "a or b or c"
    | D -> "d"
```

Matching on lists

Lists can be matched explicitly in the form `[x;y;z]` or in the "cons" form `head::tail` :

```
let y =
    match [1;2;3] with
    // binding to explicit positions
    // square brackets used!
    | [1;x;y] -> printfn "x=%A y=%A" x y

    // binding to head::tail.
    // no square brackets used!
    | 1::tail -> printfn "tail=%A" tail

    // empty list
    | [] -> printfn "empty"
```

A similar syntax is available for matching arrays exactly `[|x;y;z|]` .

It is important to understand that sequences (aka `IEnumerables`) can *not* be matched on this way directly, because they are "lazy" and meant to be accessed one element at a time. Lists and arrays, on the other hand, are fully available to be matched on.

Of these patterns, the most common one is the "cons" pattern, often used in conjunction with recursion to loop through the elements of the list.

Here are some examples of looping through lists using recursion:

```
// loop through a list and print the values
let rec loopAndPrint aList =
    match aList with
    // empty list means we're done.
    | [] ->
        printfn "empty"

    // binding to head::tail.
    | x::xs ->
        printfn "element=%A," x
        // do all over again with the
        // rest of the list
        loopAndPrint xs

//test
loopAndPrint [1..5]

// ------------------------
// loop through a list and sum the values
let rec loopAndSum aList sumSoFar =
    match aList with
    // empty list means we're done.
    | [] ->
        sumSoFar

    // binding to head::tail.
    | x::xs ->
        let newSumSoFar = sumSoFar + x
        // do all over again with the
        // rest of the list and the new sum
        loopAndSum xs newSumSoFar

//test
loopAndSum [1..5] 0
```

The second example shows how we can carry state from one iteration of the loop to the next using a special "accumulator" parameter (called `sumSoFar` in this example). This is a very common pattern.

Matching on tuples, records and unions

Pattern matching is available for all the built-in F# types. More details in the series on types.

```
// -----------------------
// Tuple pattern matching
let aTuple = (1,2)
match aTuple with
| (1,_) -> printfn "first part is 1"
| (_,2) -> printfn "second part is 2"


// -----------------------
// Record pattern matching
type Person = {First:string; Last:string}
let person = {First="john"; Last="doe"}
match person with
| {First="john"}  -> printfn "Matched John"
| _  -> printfn "Not John"

// -----------------------
// Union pattern matching
type IntOrBool= I of int | B of bool
let intOrBool = I 42
match intOrBool with
| I i  -> printfn "Int=%i" i
| B b  -> printfn "Bool=%b" b
```

Matching the whole and the part with the "as" keyword

Sometimes you want to match the individual components of the value *and* also the whole thing. You can use the `as` keyword for this.

```
let y =
    match (1,0) with
    // binding to three values
    | (x,y) as t ->
        printfn "x=%A and y=%A" x y
        printfn "The whole tuple is %A" t
```

Matching on subtypes

You can match on subtypes, using the `:?` operator, which gives you a crude polymorphism:

```
let x = new Object()
let y =
    match x with
    | :? System.Int32 ->
        printfn "matched an int"
    | :? System.DateTime ->
        printfn "matched a datetime"
    | _ ->
        printfn "another type"
```

This only works to find subclasses of a parent class (in this case, Object). The overall type of the expression has the parent class as input.

Note that in some cases, you may need to "box" the value.

```
let detectType v =
    match v with
        | :? int -> printfn "this is an int"
        | _ -> printfn "something else"
// error FS0008: This runtime coercion or type test from type 'a to int
// involves an indeterminate type based on information prior to this program point.
// Runtime type tests are not allowed on some types. Further type annotations are need
ed.
```

The message tells you the problem: "runtime type tests are not allowed on some types". The answer is to "box" the value which forces it into a reference type, and then you can type check it:

```
let detectTypeBoxed v =
    match box v with      // used "box v"
        | :? int -> printfn "this is an int"
        | _ -> printfn "something else"

//test
detectTypeBoxed 1
detectTypeBoxed 3.14
```

In my opinion, matching and dispatching on types is a code smell, just as it is in object-oriented programming. It is occasionally necessary, but used carelessly is an indication of poor design.

In a good object oriented design, the correct approach would be to use polymorphism to replace the subtype tests, along with techniques such as double dispatch. So if you are doing this kind of OO in F#, you should probably use those same techniques.

Matching on multiple values

All the patterns we've looked at so far do pattern matching on a *single* value. How can you do it for two or more?

The short answer is: you can't. Matches are only allowed on single values.

But wait a minute -- could we combine two values into a *single* tuple on the fly and match on that? Yes, we can!

```
let matchOnTwoParameters x y =
    match (x,y) with
    | (1,y) ->
        printfn "x=1 and y=%A" y
    | (x,1) ->
        printfn "x=%A and y=1" x
```

And indeed, this trick will work whenever you want to match on a set of values -- just group them all into a single tuple.

```
let matchOnTwoTuples x y =
    match (x,y) with
    | (1,_),(1,_) -> "both start with 1"
    | (_,2),(_,2) -> "both end with 2"
    | _ -> "something else"

// test
matchOnTwoTuples (1,3) (1,2)
matchOnTwoTuples (3,2) (1,2)
```

Guards, or the "when" clause

Sometimes pattern matching is just not enough, as we saw in this example:

```
let elementsAreEqual aTuple =
    match aTuple with
    | (x,y) ->
        if (x=y) then printfn "both parts are the same"
        else printfn "both parts are different"
```

Pattern matching is based on patterns only -- it can't use functions or other kinds of conditional tests.

But there *is* a way to do the equality test as part of the pattern match -- using an additional `when` clause to the left of the function arrow. These clauses are known as "guards".

Here's the same logic written using a guard instead:

```
let elementsAreEqual aTuple =
    match aTuple with
    | (x,y) when x=y ->
        printfn "both parts are the same"
    | _ ->
        printfn "both parts are different"
```

This is nicer, because we have integrated the test into the pattern proper, rather than using a test after the match has been done.

Guards can be used for all sorts of things that pure patterns can't be used for, such as:

- comparing the bound values
- testing object properties
- doing other kinds of matching, such as regular expressions
- conditionals derived from functions

Let's look at some examples of these:

```
// --------------------------------
// comparing values in a when clause
let makeOrdered aTuple =
    match aTuple with
    // swap if x is bigger than y
    | (x,y) when x > y -> (y,x)

    // otherwise leave alone
    | _ -> aTuple

//test
makeOrdered (1,2)
makeOrdered (2,1)

// --------------------------------
// testing properties in a when clause
let isAM aDate =
    match aDate:System.DateTime with
    | x when x.Hour <= 12->
        printfn "AM"

    // otherwise leave alone
    | _ ->
        printfn "PM"
```

```
//test
isAM System.DateTime.Now

// -------------------------------
// pattern matching using regular expressions
open System.Text.RegularExpressions

let classifyString aString =
    match aString with
    | x when Regex.Match(x,@".+@.+").Success->
        printfn "%s is an email" aString

    // otherwise leave alone
    | _ ->
        printfn "%s is something else" aString


//test
classifyString "alice@example.com"
classifyString "google.com"

// -------------------------------
// pattern matching using arbitrary conditionals
let fizzBuzz x =
    match x with
    | i when i % 15 = 0 ->
        printfn "fizzbuzz"
    | i when i % 3 = 0 ->
        printfn "fizz"
    | i when i % 5 = 0 ->
        printfn "buzz"
    | i  ->
        printfn "%i" i

//test
[1..30] |> List.iter fizzBuzz
```

Using active patterns instead of guards

Guards are great for one-off matches. But if there are certain guards that you use over and over, consider using active patterns instead.

For example, the email example above could be rewritten as follows:

```
open System.Text.RegularExpressions

// create an active pattern to match an email address
let (|EmailAddress|_|) input =
   let m = Regex.Match(input,@".+@.+")
   if (m.Success) then Some input else None

// use the active pattern in the match
let classifyString aString =
    match aString with
    | EmailAddress x ->
        printfn "%s is an email" x

    // otherwise leave alone
    | _ ->
        printfn "%s is something else" aString

//test
classifyString "alice@example.com"
classifyString "google.com"
```

You can see other examples of active patterns in a previous post.

The "function" keyword

In the examples so far, we've seen a lot of this:

```
let f aValue =
    match aValue with
    | _ -> "something"
```

In the special case of function definitions we can simplify this dramatically by using the `function` keyword.

```
let f =
    function
    | _ -> "something"
```

As you can see, the `aValue` parameter has completely disappeared, along with the `match..with`.

This keyword is *not* the same as the `fun` keyword for standard lambdas, rather it combines `fun` and `match..with` in a single step.

The `function` keyword works anywhere a function definition or lambda can be used, such as nested matches:

```
// using match..with
let f aValue =
    match aValue with
    | x ->
        match x with
        | _ -> "something"

// using function keyword
let f =
    function
    | x ->
        function
        | _ -> "something"
```

or lambdas passed to a higher order function:

```
// using match..with
[2..10] |> List.map (fun i ->
        match i with
        | 2 | 3 | 5 | 7 -> sprintf "%i is prime" i
        | _ -> sprintf "%i is not prime" i
        )

// using function keyword
[2..10] |> List.map (function
        | 2 | 3 | 5 | 7 -> sprintf "prime"
        | _ -> sprintf "not prime"
        )
```

A minor drawback of `function` compared with `match..with` is that you can't see the original input value and have to rely on value bindings in the pattern.

Exception handling with try..with

In the previous post, we looked at catching exceptions with the `try..with` expression.

```
try
    failwith "fail"
with
    | Failure msg -> "caught: " + msg
    | :? System.InvalidOperationException as ex -> "unexpected"
```

The `try..with` expression implements pattern matching in the same way as `match..with`.

So in the above example we see the use of matching on a custom pattern

- `| Failure msg` is an example of matching on (what looks like) an active pattern
- `| :? System.InvalidOperationException as ex` is an example of matching on the subtype (with the use of `as` as well).

Because the `try..with` expression implements full pattern matching, we can also use guards as well, if needed to add extra conditional logic:

```
let debugMode = false
try
    failwith "fail"
with
    | Failure msg when debugMode  ->
        reraise()
    | Failure msg when not debugMode ->
        printfn "silently logged in production: %s" msg
```

Wrapping match expressions with functions

Match expressions are very useful, but can lead to complex code if not used carefully.

The main problem is that match expressions doesn't compose very well. That is, it is hard to chain `match..with` expressions and build simple ones into complex ones.

The best way of avoiding this is to wrap `match..with` expressions into functions, which can then be composed nicely.

Here's a simple example. The `match x with 42` is wrapped in a `isAnswerToEverything` function.

```
let times6 x = x * 6

let isAnswerToEverything x =
    match x with
    | 42 -> (x,true)
    | _ -> (x,false)

// the function can be used for chaining or composition
[1..10] |> List.map (times6 >> isAnswerToEverything)
```

Library functions to replace explicit matching

Most built-in F# types have such functions already available.

For example, instead of using recursion to loop through lists, you should try to use the functions in the `List` module, which will do almost everything you need.

In particular, the function we wrote earlier:

```
let rec loopAndSum aList sumSoFar =
    match aList with
    | [] ->
        sumSoFar
    | x::xs ->
        let newSumSoFar = sumSoFar + x
        loopAndSum xs newSumSoFar
```

can be rewritten using the `List` module in at least three different ways!

```
// simplest
let loopAndSum1 aList = List.sum aList
[1..10] |> loopAndSum1

// reduce is very powerful
let loopAndSum2 aList = List.reduce (+) aList
[1..10] |> loopAndSum2

// fold is most powerful of all
let loopAndSum3 aList = List.fold (fun sum i -> sum+i) 0 aList
[1..10] |> loopAndSum3
```

Similarly, the Option type (discussed at length in this post) has an associated `Option` module with many useful functions.

For example, a function that does a match on `Some` vs `None` can be replaced with `Option.map`:

```
// unnecessary to implement this explicitly
let addOneIfValid optionalInt =
    match optionalInt with
    | Some i -> Some (i + 1)
    | None -> None

Some 42 |> addOneIfValid

// much easier to use the built in function
let addOneIfValid2 optionalInt =
    optionalInt |> Option.map (fun i->i+1)

Some 42 |> addOneIfValid2
```

Creating "fold" functions to hide matching logic

Finally, if you create your own types which need to be frequently matched, it is good practice to create a corresponding generic "fold" function that wraps it nicely.

For example, here is a type for defining temperature.

```
type TemperatureType  = F of float | C of float
```

Chances are, we will matching these cases a lot, so let's create a generic function that will do the matching for us.

```
module Temperature =
    let fold fahrenheitFunction celsiusFunction aTemp =
        match aTemp with
        | F f -> fahrenheitFunction f
        | C c -> celsiusFunction c
```

All `fold` functions follow this same general pattern:

- there is one function for each case in the union structure (or clause in the match pattern)
- finally, the actual value to match on comes last. (Why? See the post on "designing functions for partial application")

Now we have our fold function, we can use it in different contexts.

Let's start by testing for a fever. We need a function for testing degrees F for fever and another one for testing degrees C for fever.

And then we combine them both using the fold function.

```
let fFever tempF =
    if tempF > 100.0 then "Fever!" else "OK"

let cFever tempC =
    if tempC > 38.0 then "Fever!" else "OK"

// combine using the fold
let isFever aTemp = Temperature.fold fFever cFever aTemp
```

And now we can test.

```
let normalTemp = C 37.0
let result1 = isFever normalTemp

let highTemp = F 103.1
let result2 = isFever highTemp
```

For a completely different use, let's write a temperature conversion utility.

Again we start by writing the functions for each case, and then combine them.

```
let fConversion tempF =
    let convertedValue = (tempF - 32.0) / 1.8
    TemperatureType.C convertedValue    //wrapped in type

let cConversion tempC =
    let convertedValue = (tempC * 1.8) + 32.0
    TemperatureType.F convertedValue    //wrapped in type

// combine using the fold
let convert aTemp = Temperature.fold fConversion cConversion aTemp
```

Note that the conversion functions wrap the converted values in a new `TemperatureType`, so the `convert` function has the signature:

```
val convert : TemperatureType -> TemperatureType
```

And now we can test.

```
let c20 = C 20.0
let resultInF = convert c20

let f75 = F 75.0
let resultInC = convert f75
```

We can even call convert twice in a row, and we should get back the same temperature that we started with!

```
let resultInC = C 20.0 |> convert |> convert
```

There will be much more discussion on folds in the upcoming series on recursion and recursive types.

Formatted text using printf

In this post, we'll take a small detour and look at how to create formatted text. The printing and formatting functions are technically library functions, but in practice they as used as if they were part of the core language.

F# supports two distinct styles of formatting text:

- The standard .NET technique of "composite formatting" as seen in `String.Format`, `Console.WriteLine` and other places.
- The C-style technique of using `printf` and the associated family of functions such as `printfn`, `sprintf` and so on.

String.Format vs printf

The composite formatting technique is available in all .NET languages, and you are probably familiar with it from C#.

```
Console.WriteLine("A string: {0}. An int: {1}. A float: {2}. A bool: {3}","hello",42,3
.14,true)
```

The `printf` technique, on the other hand, is based on the C-style format strings:

```
printfn "A string: %s. An int: %i. A float: %f. A bool: %b" "hello" 42 3.14 true
```

As you have seen, the `printf` technique is very common in F#, while `String.Format`, `Console.Write` and so on, are rarely used.

Why is `printf` preferred and considered idiomatic for F#? The reasons are:

- It is statically type checked.
- It is a well-behaved F# function and so supports partial application, etc.
- It supports native F# types.

printf is statically type checked

Unlike `String.Format`, `printf` is *statically type checked*, both for the types of the parameters, and the number.

For example, here are two snippets using `printf` that will fail to compile:

```
// wrong parameter type
printfn "A string: %s" 42

// wrong number of parameters
printfn "A string: %s" "Hello" 42
```

The equivalent code using composite formatting will compile fine but either work incorrectly but silently, or give a runtime error:

```
// wrong parameter type
Console.WriteLine("A string: {0}", 42)   //works!

// wrong number of parameters
Console.WriteLine("A string: {0}","Hello",42)  //works!
Console.WriteLine("A string: {0}. An int: {1}","Hello") //FormatException
```

printf supports partial application

The .NET formatting functions require all parameters to be passed in *at the same time*.

But `printf` is a standard, well-behaved F# function, and so supports partial application.

Here are some examples:

```
// partial application - explicit parameters
let printStringAndInt s i =  printfn "A string: %s. An int: %i" s i
let printHelloAndInt i = printStringAndInt "Hello" i
do printHelloAndInt 42

// partial application - point free style
let printInt =  printfn "An int: %i"
do printInt 42
```

And of course, `printf` can be used for function parameters anywhere a standard function can be used.

```
let doSomething printerFn x y =
    let result = x + y
    printerFn "result is" result

let callback = printfn "%s %i"
do doSomething callback 3 4
```

This also includes the higher order functions for lists, etc:

```
[1..5] |> List.map (sprintf "i=%i")
```

printf supports native F# types

For non-primitive types, the .NET formatting functions only support using `ToString()`, but `printf` supports native F# types using the `%A` specifier:

```
// tuple printing
let t = (1,2)
Console.WriteLine("A tuple: {0}", t)
printfn "A tuple: %A" t

// record printing
type Person = {First:string; Last:string}
let johnDoe = {First="John"; Last="Doe"}
Console.WriteLine("A record: {0}", johnDoe )
printfn "A record: %A" johnDoe

// union types printing
type Temperature = F of int | C of int
let freezing = F 32
Console.WriteLine("A union: {0}", freezing )
printfn "A union: %A" freezing
```

As you can see, tuple types have a nice `ToString()` but other user defined types don't, so if you want to use them with the .NET formatting functions, you will have to override the `ToString()` method explicitly.

printf gotchas

There are a couple of "gotchas" to be aware of when using `printf`.

First, if there are *too few* parameters, rather than too many, the compiler will *not* complain immediately, but might give cryptic errors later.

```
// too few parameters
printfn "A string: %s An int: %i" "Hello"
```

The reason, of course, is that this is not an error at all; `printf` is just being partially applied! See the discussion of partial application if you are not clear of why this happens.

Another issue is that the "format strings" are not actually strings.

In the .NET formatting model, the formatting strings are normal strings, so you can pass them around, store them in resource files, and so on. Which means that the following code works fine:

```
let netFormatString = "A string: {0}"
Console.WriteLine(netFormatString, "hello")
```

On the other hand, the "format strings" that are the first argument to `printf` are not really strings at all, but something called a `TextWriterFormat`. Which means that the following code does **not** work:

```
let fsharpFormatString = "A string: %s"
printfn fsharpFormatString  "Hello"
```

The compiler does some magic behind the scenes to convert the string constant `"A string: %s"` into the appropriate TextWriterFormat. The TextWriterFormat is the key component that "knows" the type of the format string, such as `string->unit` or `string->int->unit`, which in turn allows `printf` to be typesafe.

If you want to emulate the compiler, you can create your own TextWriterFormat value from a string using the `Printf.TextWriterFormat` type in the `Microsoft.FSharp.Core.Printf` module.

If the format string is "inline", the compiler can deduce the type for you during binding:

```
let format:Printf.TextWriterFormat<_> = "A string: %s"
printfn format "Hello"
```

But if the format string is truly dynamic (e.g. stored in a resource or created on the fly), the compiler cannot deduce the type for you, and you must explicitly provide it with the constructor.

In the example below, my first format string has a single string parameter and returns a unit, so I have to specify `string->unit` as the format type. And in the second case, I have to specify `string->int->unit` as the format type.

```
let formatAString = "A string: %s"
let formatAStringAndInt = "A string: %s. An int: %i"

//convert to TextWriterFormat
let twFormat1  = Printf.TextWriterFormat<string->unit>(formatAString)
printfn twFormat1 "Hello"
let twFormat2  = Printf.TextWriterFormat<string->int->unit>(formatAStringAndInt)
printfn twFormat2  "Hello" 42
```

I won't go into detail on exactly how `printf and` TextWriterFormat` work together right now -- just be aware that is not just a matter of simple format strings being passed around.

Finally, it's worth noting that `printf` and family are *not* thread-safe, while `Console.Write` and family *are*.

How to specify a format

The "%" format specifications are quite similar to those used in C, but with some special customizations for F#.

As with C, the characters immediately following the `%` have a specific meaning, as shown below.

```
%[flags][width][.precision]specifier
```

We'll discuss each of these attributes in more detail below.

Formatting for dummies

The most commonly used format specifiers are:

- `%s` for strings
- `%b` for bools
- `%i` for ints
- `%f` for floats
- `%A` for pretty-printing tuples, records and union types
- `%O` for other objects, using `ToString()`

These six will probably meet most of your basic needs.

Escaping %

The `%` character on its own will cause an error. To escape it, just double it up:

```
printfn "unescaped: %" // error
printfn "escape: %%"
```

Controlling width and alignment

When formatting fixed width columns and tables, you need to have control of the alignment and width.

You can do that with the "width" and "flags" options.

- `%5s` , `%5i` . A number sets the width of the value
- `%*s` , `%*i` . A star sets the width of the value dynamically (from an extra parameter just before the param to format)
- `%-s` , `%-i` . A hyphen left justifies the value.

Here are some examples of these in use:

```
let rows = [ (1,"a"); (-22,"bb"); (333,"ccc"); (-4444,"dddd") ]

// no alignment
for (i,s) in rows do
    printfn "|%i|%s|" i s

// with alignment
for (i,s) in rows do
    printfn "|%5i|%5s|" i s

// with left alignment for column 2
for (i,s) in rows do
    printfn "|%5i|%-5s|" i s

// with dynamic column width=20 for column 1
for (i,s) in rows do
    printfn "|%*i|%-5s|" 20 i s

// with dynamic column width for column 1 and column 2
for (i,s) in rows do
    printfn "|%*i|%-*s|" 20 i 10 s
```

Formatting integers

There are some special options for basic integer types:

- `%i` or `%d` for signed ints
- `%u` for unsigned ints
- `%x` and `%X` for lowercase and uppercase hex
- `%o` for octal

Here are some examples:

```
printfn "signed8: %i unsigned8: %u" -1y -1y
printfn "signed16: %i unsigned16: %u" -1s -1s
printfn "signed32: %i unsigned32: %u" -1 -1
printfn "signed64: %i unsigned64: %u" -1L -1L
printfn "uppercase hex: %X lowercase hex: %x octal: %o" 255 255 255
printfn "byte: %i " 'A'B
```

The specifiers do not enforce any type safety within the integer types. As you can see from the examples above, you can pass a signed int to an unsigned specifier without problems. What is different is how it is formatted. The unsigned specifiers treat the int as unsigned no matter how it is actually typed.

Note that `BigInteger` is *not* a basic integer type, so you must format it with `%A` or `%O`.

```
printfn "bigInt: %i " 123456789I  // Error
printfn "bigInt: %A " 123456789I  // OK
```

You can control the formatting of signs and zero padding using the flags:

- `%0i` pads with zeros
- `%+i` shows a plus sign
- `% i` shows a blank in place of a plus sign

Here are some examples:

```
let rows = [ (1,"a"); (-22,"bb"); (333,"ccc"); (-4444,"dddd") ]

// with alignment
for (i,s) in rows do
    printfn "|%5i|%5s|" i s

// with plus signs
for (i,s) in rows do
    printfn "|%+5i|%5s|" i s

// with zero pad
for (i,s) in rows do
    printfn "|%0+5i|%5s|" i s

// with left align
for (i,s) in rows do
    printfn "|%-5i|%5s|" i s

// with left align and plus
for (i,s) in rows do
    printfn "|%+-5i|%5s|" i s

// with left align and space instead of plus
for (i,s) in rows do
    printfn "|% -5i|%5s|" i s
```

Formatting floats and decimals

For floating point types, there are also some special options:

- `%f` for standard format
- `%e` or `%E` for exponential format
- `%g` or `%G` for the more compact of `f` and `e`.
- `%M` for decimals

Here are some examples:

```
let pi = 3.14
printfn "float: %f exponent: %e compact: %g" pi pi pi

let petabyte = pown 2.0 50
printfn "float: %f exponent: %e compact: %g" petabyte petabyte petabyte
```

The decimal type can be used with the floating point specifiers, but you might lose some precision. The `%M` specifier can be used to ensure that no precision is lost. You can see the difference with this example:

```
let largeM = 123456789.123456789M  // a decimal
printfn "float: %f decimal: %M" largeM largeM
```

You can control the precision of floats using a precision specification, such as `%.2f` and `%.4f`. For the `%f` and `%e` specifiers, the precision affects the number of digits after the decimal point, while for `%g` it is the number of digits in total. Here's an example:

```
printfn "2 digits precision: %.2f. 4 digits precision: %.4f." 123.456789 123.456789
// output => 2 digits precision: 123.46. 4 digits precision: 123.4568.
printfn "2 digits precision: %.2e. 4 digits precision: %.4e." 123.456789 123.456789
// output => 2 digits precision: 1.23e+002. 4 digits precision: 1.2346e+002.
printfn "2 digits precision: %.2g. 4 digits precision: %.4g." 123.456789 123.456789
// output => 2 digits precision: 1.2e+02. 4 digits precision: 123.5.
```

The alignment and width flags work for floats and decimals as well.

```
printfn "|%f|" pi     // normal
printfn "|%10f|" pi   // width
printfn "|%010f|" pi  // zero-pad
printfn "|%-10f|" pi  // left aligned
printfn "|%0-10f|" pi // left zero-pad
```

Custom formatting functions

There are two special format specifiers that allow to you pass in a function rather than just a simple value.

- `%t` expects a function that outputs some text with no input
- `%a` expects a function that outputs some text from a given input

Here's an example of using `%t`:

```
open System.IO

//define the function
let printHello (tw:TextWriter) = tw.Write("hello")

//test it
printfn "custom function: %t" printHello
```

Obviously, since the callback function takes no parameters, it will probably be a closure that does reference some other value. Here's an example that prints random numbers:

```
open System
open System.IO

//define the function using a closure
let printRand =
    let rand = new Random()
    // return the actual printing function
    fun (tw:TextWriter) -> tw.Write(rand.Next(1,100))

//test it
for i in [1..5] do
    printfn "rand = %t" printRand
```

For the `%a` specifier, the callback function takes an extra parameter. That is, when using the `%a` specifier, you must pass in both a function and a value to format.

Here's an example of custom formatting a tuple:

```
open System
open System.IO

//define the callback function
//note that the data parameter comes after the TextWriter
let printLatLong (tw:TextWriter) (lat,long) =
    tw.Write("lat:{0} long:{1}", lat, long)

// test it
let latLongs = [ (1,2); (3,4); (5,6)]
for latLong  in latLongs  do
    // function and value both passed in to printfn
    printfn "latLong = %a" printLatLong latLong
```

Date formatting

There are no special format specifiers for dates in F#.

If you want to format dates, you have a couple of options:

- Use `ToString` to convert the date into a string, and then use the `%s` specifier
- Use a custom callback function with the `%a` specifier as described above

Here are the two approaches in use:

```
// function to format a date
let yymmdd1 (date:DateTime) = date.ToString("yy.MM.dd")

// function to format a date onto a TextWriter
let yymmdd2 (tw:TextWriter) (date:DateTime) = tw.Write("{0:yy.MM.dd}", date)

// test it
for i in [1..5] do
    let date = DateTime.Now.AddDays(float i)

    // using %s
    printfn "using ToString = %s" (yymmdd1 date)

    // using %a
    printfn "using a callback = %a" yymmdd2 date
```

Which approach is better?

The `ToString` with `%s` is easier to test and use, but it will be less efficient than writing directly to a TextWriter.

The printf family of functions

There are a number of variants of `printf` functions. Here is a quick guide:

F# function	C# equivalent	Comment
`printf` and `printfn`	`Console.Write` and `Console.WriteLine`	Functions starting with "print" write to standard out.
`eprintf` and `eprintfn`	`Console.Error.Write` and `Console.Error.WriteLine`	Functions starting with "eprint" write to standard error.
`fprintf` and `fprintfn`	`TextWriter.Write` and `TextWriter.WriteLine`	Functions starting with "fprint" write to a TextWriter.
`sprintf`	`String.Format`	Functions starting with "sprint" return a string.
`bprintf`	`StringBuilder.AppendFormat`	Functions starting with "bprint" write to a StringBuilder.
`kprintf`, `kfprintf`, `ksprintf` and `kbprintf`	No equivalent	Functions that accept a continuation. See next section for a discussion.

All of these except `bprintf` and the `kXXX` family are automatically available (via Microsoft.FSharp.Core.ExtraTopLevelOperators). But if you need to access them using a module, they are in the `Printf` module.

The usage of these should be obvious (except for the `kxxx` family, of which more below).

A particularly useful technique is to use partial application to "bake in" a TextWriter or StringBuilder.

Here is an example using a StringBuilder:

```
let printToSb s i =
    let sb = new System.Text.StringBuilder()

    // use partial application to fix the StringBuilder
    let myPrint format = Printf.bprintf sb format

    do myPrint "A string: %s. " s
    do myPrint "An int: %i" i

    //get the result
    sb.ToString()

// test
printToSb "hello" 42
```

And here is an example using a TextWriter:

```
open System
open System.IO

let printToFile filename s i =
    let myDocsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)
    let fullPath = Path.Combine(myDocsPath, filename)
    use sw = new StreamWriter(path=fullPath)

    // use partial application to fix the TextWriter
    let myPrint format = fprintf sw format

    do myPrint "A string: %s. " s
    do myPrint "An int: %i" i

    //get the result
    sw.Close()

// test
printToFile "myfile.txt" "hello" 42
```

More on partially applying printf

Note that in both cases above, we had to pass a format parameter when creating the partial application.

That is, we had to do:

```
let myPrint format = fprintf sw format
```

rather than the point-free version:

```
let myPrint  = fprintf sw
```

This stops the compiler complaining about an incorrect type. The reason why is non-obvious. We briefly mentioned the `TextWriterFormat` above as the first parameter to `printf`. It turns out that `printf` is not actually a particular function, like `String.Format`, but rather a generic function that has to be parameterized with a TextWriterFormat (or the similar StringFormat) in order to become "real".

So, to be safe, it is best to always pair a `printf` with a format parameter, rather than being overly aggressive with the partial application.

The kprintf functions

The four `kXXX` functions are similar to their cousins, except that they take an extra parameter -- a continuation. That is, a function to be called immediately after the formatting has been done.

Here's a simple snippet:

```
let doAfter s =
    printfn "Done"
    // return the result
    s

let result = Printf.ksprintf doAfter "%s" "Hello"
```

Why would you want this? A number of reasons:

- You can pass the result to another function that does something useful, such as a logging framework
- You can do things such as flushing the TextWriter
- You can raise an event

Let's look at a sample that uses a external logging framework plus custom events.

First, let's create a simple logging class along the lines of log4net or System.Diagnostics.Trace. In practice, this would be replaced by a real third-party library.

```
open System
open System.IO

// a logging library such as log4net
// or System.Diagnostics.Trace
type Logger(name) =

    let currentTime (tw:TextWriter) =
        tw.Write("{0:s}",DateTime.Now)

    let logEvent level msg =
        printfn "%t %s [%s] %s" currentTime level name msg

    member this.LogInfo msg =
        logEvent "INFO" msg

    member this.LogError msg =
        logEvent "ERROR" msg

    static member CreateLogger name =
        new Logger(name)
```

Next in my application code, I do the following:

- Create an instance of the logging framework. I've hard-coded the factory method here, but you could also use an IoC container.
- Create helper functions called `logInfo` and `logError` that call the logging framework, and in the case of `logError`, show a popup message as well.

```
// my application code
module MyApplication =

    let logger = Logger.CreateLogger("MyApp")

    // create a logInfo using the Logger class
    let logInfo format =
        let doAfter s =
            logger.LogInfo(s)
        Printf.ksprintf doAfter format

    // create a logError using the Logger class
    let logError format =
        let doAfter s =
            logger.LogError(s)
            System.Windows.Forms.MessageBox.Show(s) |> ignore
        Printf.ksprintf doAfter format

    // function to exercise the logging
    let test() =
        do logInfo "Message #%i" 1
        do logInfo "Message #%i" 2
        do logError "Oops! an error occurred in my app"
```

Finally, when we run the `test` function, we should get the message written to the console, and also see the popup message:

```
MyApplication.test()
```

You could also create an object-oriented version of the helper methods by creating a "FormattingLogger" wrapper class around the logging library, as shown below.

```
type FormattingLogger(name) =

    let logger = Logger.CreateLogger(name)

    // create a logInfo using the Logger class
    member this.logInfo format =
        let doAfter s =
            logger.LogInfo(s)
        Printf.ksprintf doAfter format

    // create a logError using the Logger class
    member this.logError format =
        let doAfter s =
            logger.LogError(s)
            System.Windows.Forms.MessageBox.Show(s) |> ignore
        Printf.ksprintf doAfter format

    static member createLogger name =
        new FormattingLogger(name)

// my application code
module MyApplication2 =

    let logger = FormattingLogger.createLogger("MyApp2")

    let test() =
        do logger.logInfo "Message #%i" 1
        do logger.logInfo "Message #%i" 2
        do logger.logError "Oops! an error occurred in app 2"

// test
MyApplication2.test()
```

The object-oriented approach, although more familiar, is not automatically better! The pros and cons of OO methods vs. pure functions are discussed here.

Worked example: Parsing command line arguments

Now that we've seen how the match expression works, let's look at some examples in practice. But first, a word about the design approach.

Application design in F#

We've seen that a generic function takes input and emits output. But in a sense, that approach applies at *any* level of functional code, even at the top level.

In fact, we can say that a functional *application* takes input, transforms it, and emits output:



Now ideally, the transformations work within the pure type-safe world that we create to model the domain, but unfortunately, the real world is untyped! That is, the input is likely to be simple strings or bytes, and the output also.

How can we work with this? The obvious solution is to have a separate stage to convert the input to our pure internal model, and then another separate stage to convert from the internal model to the output.



In this way, we can hide the messiness of the real world from the core of the application. This "keep your model pure" approach is similar to the "Hexagonal Architecture" concept in the large, or the MVC pattern in the small.

In this post and the next, we'll see some simple examples of this.

Example: parsing a command line

We talked about the match expression in general in the previous post, so let's look at a real example where it is useful, namely parsing a command line.

We'll design and implement two slightly different versions, one with a basic internal model, and second one with some improvements.

Requirements

Let's say that we have three commandline options: "verbose", "subdirectories", and "orderby". "Verbose" and "subdirectories" are flags, while "orderby" has two choices: "by size" and "by name".

So the command line params would look like

```
MYAPP [/V] [/S] [/O order]
/V    verbose
/S    include subdirectories
/O    order by. Parameter is one of
        N - order by name.
        S - order by size
```

First version

Following the design rule above, we can see that:

- the input will be an array (or list) of strings, one for each argument.
- the internal model will be a set of types that model the (tiny) domain.
- the output is out of scope in this example.

So we'll start by first creating the internal model of the parameters, and then look at how we can parse the input into types used in the internal model.

Here's a first stab at the model:

```
// constants used later
let OrderByName = "N"
let OrderBySize = "S"

// set up a type to represent the options
type CommandLineOptions = {
    verbose: bool;
    subdirectories: bool;
    orderby: string;
    }
```

Ok, that looks alright. Now let's parse the arguments.

The parsing logic is very similar to the `loopAndSum` example in the previous post.

- We create a recursive loop on the list of arguments.
- Each time through the loop, we parse one argument.
- The options parsed so far are passed into each loop as a parameter (the "accumulator" pattern).

```
let rec parseCommandLine args optionsSoFar =
    match args with
    // empty list means we're done.
    | [] ->
        optionsSoFar

    // match verbose flag
    | "/v"::xs ->
        let newOptionsSoFar = { optionsSoFar with verbose=true}
        parseCommandLine xs newOptionsSoFar

    // match subdirectories flag
    | "/s"::xs ->
        let newOptionsSoFar = { optionsSoFar with subdirectories=true}
        parseCommandLine xs newOptionsSoFar

    // match orderBy by flag
    | "/o"::xs ->
        //start a submatch on the next arg
        match xs with
        | "S"::xss ->
            let newOptionsSoFar = { optionsSoFar with orderby=OrderBySize}
            parseCommandLine xss newOptionsSoFar

        | "N"::xss ->
            let newOptionsSoFar = { optionsSoFar with orderby=OrderByName}
            parseCommandLine xss newOptionsSoFar

        // handle unrecognized option and keep looping
        | _ ->
            eprintfn "OrderBy needs a second argument"
            parseCommandLine xs optionsSoFar

    // handle unrecognized option and keep looping
    | x::xs ->
        eprintfn "Option '%s' is unrecognized" x
        parseCommandLine xs optionsSoFar
```

This code is straightforward, I hope.

Each match consist of a `option::restOfList` pattern. If the option is matched, a new `optionsSoFar` value is created and the loop repeats with the remaining list, until the list becomes empty, at which point we can exit the loop and return the `optionsSoFar` value as the final result.

There are two special cases:

- Matching the "orderBy" option creates a submatch pattern that looks at the first item in the rest of the list and if not found, complains about a missing second parameter.
- The very last match on the main `match..with` is not a wildcard, but a "bind to value". Just like a wildcard, this will always succeed, but because we havd bound to the value, it allows us to print the offending unmatched argument.
- Note that for printing errors, we use `eprintf` rather than `printf`. This will write to STDERR rather than STDOUT.

So now let's test this:

```
parseCommandLine ["/v"; "/s"]
```

Oops! That didn't work -- we need to pass in an initial `optionsSoFar` argument! Lets try again:

```
// define the defaults to pass in
let defaultOptions = {
    verbose = false;
    subdirectories = false;
    orderby = ByName
    }

// test it
parseCommandLine ["/v"] defaultOptions
parseCommandLine ["/v"; "/s"] defaultOptions
parseCommandLine ["/o"; "S"] defaultOptions
```

Check that the output is what you would expect.

And we should also check the error cases:

```
parseCommandLine ["/v"; "xyz"] defaultOptions
parseCommandLine ["/o"; "xyz"] defaultOptions
```

You should see the error messages in these cases now.

Before we finish this implementation, let's fix something annoying. We are passing in these default options every time -- can we get rid of them?

This is a very common situation: you have a recursive function that takes a "accumulator" parameter, but you don't want to be passing initial values all the time.

The answer is simple: just create another function that calls the recursive function with the defaults.

Normally, this second one is the "public" one and the recursive one is hidden, so we will rewrite the code as follows:

- Rename `parseCommandLine` to `parseCommandLineRec`. There are other naming conventions you could use as well, such as `parseCommandLine'` with a tick mark, or `innerParseCommandLine`.
- Create a new `parseCommandLine` that calls `parseCommandLineRec` with the defaults

```
// create the "helper" recursive function
let rec parseCommandLineRec args optionsSoFar =
    // implementation as above

// create the "public" parse function
let parseCommandLine args =
    // create the defaults
    let defaultOptions = {
        verbose = false;
        subdirectories = false;
        orderby = OrderByName
        }

    // call the recursive one with the initial options
    parseCommandLineRec args defaultOptions
```

In this case the helper function can stand on its own. But if you really want to hide it, you can put it as a nested subfunction in the defintion of `parseCommandLine` itself.

```
// create the "public" parse function
let parseCommandLine args =
    // create the defaults
    let defaultOptions =
        // implementation as above

    // inner recursive function
    let rec parseCommandLineRec args optionsSoFar =
        // implementation as above

    // call the recursive one with the initial options
    parseCommandLineRec args defaultOptions
```

In this case, I think it would just make things more complicated, so I have kept them separate.

So, here is all the code at once, wrapped in a module:

```
module CommandLineV1 =

    // constants used later
    let OrderByName = "N"
    let OrderBySize = "S"

    // set up a type to represent the options
    type CommandLineOptions = {
        verbose: bool;
        subdirectories: bool;
        orderby: string;
        }

    // create the "helper" recursive function
    let rec parseCommandLineRec args optionsSoFar =
        match args with
        // empty list means we're done.
        | [] ->
            optionsSoFar

        // match verbose flag
        | "/v"::xs ->
            let newOptionsSoFar = { optionsSoFar with verbose=true}
            parseCommandLineRec xs newOptionsSoFar

        // match subdirectories flag
        | "/s"::xs ->
            let newOptionsSoFar = { optionsSoFar with subdirectories=true}
            parseCommandLineRec xs newOptionsSoFar

        // match orderBy by flag
        | "/o"::xs ->
            //start a submatch on the next arg
            match xs with
            | "S"::xss ->
                let newOptionsSoFar = { optionsSoFar with orderby=OrderBySize}
                parseCommandLineRec xss newOptionsSoFar

            | "N"::xss ->
                let newOptionsSoFar = { optionsSoFar with orderby=OrderByName}
                parseCommandLineRec xss newOptionsSoFar

            // handle unrecognized option and keep looping
            | _ ->
                eprintfn "OrderBy needs a second argument"
                parseCommandLineRec xs optionsSoFar
```

```
        // handle unrecognized option and keep looping
        | x::xs ->
            eprintfn "Option '%s' is unrecognized" x
            parseCommandLineRec xs optionsSoFar

    // create the "public" parse function
    let parseCommandLine args =
        // create the defaults
        let defaultOptions = {
            verbose = false;
            subdirectories = false;
            orderby = OrderByName
            }

        // call the recursive one with the initial options
        parseCommandLineRec args defaultOptions


// happy path
CommandLineV1.parseCommandLine ["/v"]
CommandLineV1.parseCommandLine  ["/v"; "/s"]
CommandLineV1.parseCommandLine  ["/o"; "S"]

// error handling
CommandLineV1.parseCommandLine ["/v"; "xyz"]
CommandLineV1.parseCommandLine ["/o"; "xyz"]
```

Second version

In our initial model we used bool and string to represent the possible values.

```
type CommandLineOptions = {
    verbose: bool;
    subdirectories: bool;
    orderby: string;
    }
```

There are two problems with this:

- **It doesn't *really* represent the domain.** For example, can `orderby` really be *any* string? Would my code break if I set it to "ABC"?
- **The values are not self documenting.** For example, the verbose value is a bool. We only know that the bool represents the "verbose" option because of the *context* (the field named `verbose`) it is found in. If we passed that bool around, and took it out of context, we would not know what it represented. I'm sure we have all seen C# functions with many boolean parameters like this:

```
myObject.SetUpComplicatedOptions(true,false,true,false,false);
```

Because the bool doesn't represent anything at the domain level, it is very easy to make mistakes.

The solution to both these problems is to be as specific as possible when defining the domain, typically by creating lots of very specific types.

So here's a new version of `CommandLineOptions` :

```
type OrderByOption = OrderBySize | OrderByName
type SubdirectoryOption = IncludeSubdirectories | ExcludeSubdirectories
type VerboseOption = VerboseOutput | TerseOutput

type CommandLineOptions = {
    verbose: VerboseOption;
    subdirectories: SubdirectoryOption;
    orderby: OrderByOption
    }
```

A couple of things to notice:

- There are no bools or strings anywhere.
- The names are quite explicit. This acts as documentation when a value is taken in isolation, but also means that the name is unique, which helps type inference, which in turn helps you avoid explicit type annotations.

Once we have made the changes to the domain, it is easy to fix up the parsing logic.

So, here is all the revised code, wrapped in a "v2" module:

```
module CommandLineV2 =

    type OrderByOption = OrderBySize | OrderByName
    type SubdirectoryOption = IncludeSubdirectories | ExcludeSubdirectories
    type VerboseOption = VerboseOutput | TerseOutput

    type CommandLineOptions = {
        verbose: VerboseOption;
        subdirectories: SubdirectoryOption;
        orderby: OrderByOption
        }

    // create the "helper" recursive function
    let rec parseCommandLineRec args optionsSoFar =
        match args with
        // empty list means we're done.
        | [] ->
```

```
            optionsSoFar

        // match verbose flag
        | "/v"::xs ->
            let newOptionsSoFar = { optionsSoFar with verbose=VerboseOutput}
            parseCommandLineRec xs newOptionsSoFar

        // match subdirectories flag
        | "/s"::xs ->
            let newOptionsSoFar = { optionsSoFar with subdirectories=IncludeSubdirecto
ries}
            parseCommandLineRec xs newOptionsSoFar

        // match sort order flag
        | "/o"::xs ->
            //start a submatch on the next arg
            match xs with
            | "S"::xss ->
                let newOptionsSoFar = { optionsSoFar with orderby=OrderBySize}
                parseCommandLineRec xss newOptionsSoFar
            | "N"::xss ->
                let newOptionsSoFar = { optionsSoFar with orderby=OrderByName}
                parseCommandLineRec xss newOptionsSoFar
            // handle unrecognized option and keep looping
            | _ ->
                printfn "OrderBy needs a second argument"
                parseCommandLineRec xs optionsSoFar

        // handle unrecognized option and keep looping
        | x::xs ->
            printfn "Option '%s' is unrecognized" x
            parseCommandLineRec xs optionsSoFar

    // create the "public" parse function
    let parseCommandLine args =
        // create the defaults
        let defaultOptions = {
            verbose = TerseOutput;
            subdirectories = ExcludeSubdirectories;
            orderby = OrderByName
            }

        // call the recursive one with the initial options
        parseCommandLineRec args defaultOptions

// ==============================
// tests

// happy path
CommandLineV2.parseCommandLine ["/v"]
CommandLineV2.parseCommandLine ["/v"; "/s"]
CommandLineV2.parseCommandLine ["/o"; "S"]
```

```
// error handling
CommandLineV2.parseCommandLine ["/v"; "xyz"]
CommandLineV2.parseCommandLine ["/o"; "xyz"]
```

Using fold instead of recursion?

We said in the previous post that it is good to avoid recursion where possible and use the built in functions in the `List` module like `map` and `fold` .

So can we take this advice here and fix up this code to do this?

Unfortunately, not easily. The problem is that the list functions generally work on one element at a time, while the "orderby" option requires a "lookahead" argument as well.

To make this work with something like `fold` , we need to create a "parse mode" flag to indicate whether we are in lookahead mode or not. This is possible, but I think it just adds extra complexity compared to the straightforward recursive version above.

And in a real-world situation, anything more complicated than this would be a signal that you need to switch to a proper parsing system such as FParsec.

However, just to show you it can be done with `fold` :

```
module CommandLineV3 =

    type OrderByOption = OrderBySize | OrderByName
    type SubdirectoryOption = IncludeSubdirectories | ExcludeSubdirectories
    type VerboseOption = VerboseOutput | TerseOutput

    type CommandLineOptions = {
        verbose: VerboseOption;
        subdirectories: SubdirectoryOption;
        orderby: OrderByOption
        }

    type ParseMode = TopLevel | OrderBy

    type FoldState = {
        options: CommandLineOptions ;
        parseMode: ParseMode;
        }

    // parse the top-level arguments
    // return a new FoldState
    let parseTopLevel arg optionsSoFar =
        match arg with

        // match verbose flag
```

```
        | "/v" ->
            let newOptionsSoFar = {optionsSoFar with verbose=VerboseOutput}
            {options=newOptionsSoFar; parseMode=TopLevel}

        // match subdirectories flag
        | "/s"->
            let newOptionsSoFar = { optionsSoFar with subdirectories=IncludeSubdirecto
ries}
            {options=newOptionsSoFar; parseMode=TopLevel}

        // match sort order flag
        | "/o" ->
            {options=optionsSoFar; parseMode=OrderBy}

        // handle unrecognized option and keep looping
        | x ->
            printfn "Option '%s' is unrecognized" x
            {options=optionsSoFar; parseMode=TopLevel}

    // parse the orderBy arguments
    // return a new FoldState
    let parseOrderBy arg optionsSoFar =
        match arg with
        | "S" ->
            let newOptionsSoFar = { optionsSoFar with orderby=OrderBySize}
            {options=newOptionsSoFar; parseMode=TopLevel}
        | "N" ->
            let newOptionsSoFar = { optionsSoFar with orderby=OrderByName}
            {options=newOptionsSoFar; parseMode=TopLevel}
        // handle unrecognized option and keep looping
        | _ ->
            printfn "OrderBy needs a second argument"
            {options=optionsSoFar; parseMode=TopLevel}

    // create a helper fold function
    let foldFunction state element  =
        match state with
        | {options=optionsSoFar; parseMode=TopLevel} ->
            // return new state
            parseTopLevel element optionsSoFar

        | {options=optionsSoFar; parseMode=OrderBy} ->
            // return new state
            parseOrderBy element optionsSoFar

    // create the "public" parse function
    let parseCommandLine args =

        let defaultOptions = {
            verbose = TerseOutput;
            subdirectories = ExcludeSubdirectories;
            orderby = OrderByName
            }
```

```
        let initialFoldState =
            {options=defaultOptions; parseMode=TopLevel}

        // call fold with the initial state
        args |> List.fold foldFunction initialFoldState

// ==============================
// tests

// happy path
CommandLineV3.parseCommandLine ["/v"]
CommandLineV3.parseCommandLine ["/v"; "/s"]
CommandLineV3.parseCommandLine ["/o"; "S"]

// error handling
CommandLineV3.parseCommandLine ["/v"; "xyz"]
CommandLineV3.parseCommandLine ["/o"; "xyz"]
```

By the way, can you see a subtle change of behavior in this version?

In the previous versions, if there was no parameter to the "orderBy" option, the recursive loop would still parse it next time. But in the 'fold' version, this token is swallowed and lost.

To see this, compare the two implementations:

```
// verbose set
CommandLineV2.parseCommandLine ["/o"; "/v"]

// verbose not set!
CommandLineV3.parseCommandLine ["/o"; "/v"]
```

To fix this would be even more work. Again this argues for the second implementation as the easiest to debug and maintain.

Summary

In this post we've seen how to apply pattern matching to a real-world example.

More importantly, we've seen how easy it is to create a properly designed internal model for even the smallest domain. And that this internal model provides more type safety and documentation than using primitive types such as string and bool.

In the next example, we'll do even more pattern matching!

Worked example: Roman numerals

Last time we looked at parsing a command line. This time we'll we'll look at another pattern matching example, this time using Roman numerals.

As before, we will try to have a "pure" internal model with separate stages to convert the input to the internal model, and then another separate stage to convert from the internal model to the output.



Requirements

Let's start with the requirements:

```
1) Accept a string of letters like "MMMXCLXXIV" as a string and convert it to an integ
er.
The conversions are: I=1; V=5; X=10; L=50; C=100; D=500; and M=1000;

If a lower letter comes before a higher one, the value of the higher is reduced accord
ingly, so
IV=4; IX=9; XC=90; and so on.

2) As an additional step, validate the string of letters to see if it is a valid numbe
r. For example: "IIVVMM" is a not a valid roman numeral.
```

First version

As before we'll start by first creating the internal model, and then look at how we can parse the input into the internal model.

Here's a first stab at the model. We'll treat a `RomanNumeral` as a list of `RomanDigits`.

```
type RomanDigit = int
type RomanNumeral = RomanDigit list
```

No, stop right there! A `RomanDigit` is not just *any* digit, it has to be taken from a limited set.

Also `RomanNumeral` should not just be a type alias for a list of digits. It would be better if it was its own special type as well. We can do this by creating a single case union type.

Here's a much better version:

```
type RomanDigit = I | V | X | L | C | D | M
type RomanNumeral = RomanNumeral of RomanDigit list
```

Output: Converting a numeral to an int

Now let's do the output logic, converting a Roman numeral to an int.

The digit conversion is easy:

```
/// Converts a single RomanDigit to an integer
let digitToInt =
    function
    | I -> 1
    | V -> 5
    | X -> 10
    | L -> 50
    | C -> 100
    | D -> 500
    | M -> 1000

// tests
I  |> digitToInt
V  |> digitToInt
M  |> digitToInt
```

Note that we're using the `function` keyword instead of the `match..with` expression.

To convert a list of digits, we'll use a recursive loop again. There is a special case when we need to look ahead to the next digit, and if it is larger than the current one, then use their difference.

```
let rec digitsToInt =
    function

    // empty is 0
    | [] -> 0

    // special case when a smaller comes before larger
    // convert both digits and add the difference to the sum
    // Example: "IV" and "CM"
    | smaller::larger::ns when smaller < larger ->
        (digitToInt larger - digitToInt smaller)  + digitsToInt ns

    // otherwise convert the digit and add to the sum
    | digit::ns ->
        digitToInt digit + digitsToInt ns

// tests
[I;I;I;I]  |> digitsToInt
[I;V]  |> digitsToInt
[V;I]  |> digitsToInt
[I;X]  |> digitsToInt
[M;C;M;L;X;X;I;X]  |> digitsToInt // 1979
[M;C;M;X;L;I;V] |> digitsToInt // 1944
```

Note that the "less than" operation did not have to be defined. The types are automatically sorted by their order of declaration.

Finally, we can convert the `RomanNumeral` type itself by unpacking the contents into a list and calling `digitsToInt`.

```
/// converts a RomanNumeral to an integer
let toInt (RomanNumeral digits) = digitsToInt digits

// test
let x = RomanNumeral [I;I;I;I]
x |> toInt

let x = RomanNumeral [M;C;M;L;X;X;I;X]
x |> toInt
```

That takes care of the output.

Input: Converting a string to an Roman Numeral

Now let's do the input logic, converting a string to our internal model.

First, let's handle a single character conversion. It seems straightforward.

```
let charToRomanDigit =
    function
    | 'I' -> I
    | 'V' -> V
    | 'X' -> X
    | 'L' -> L
    | 'C' -> C
    | 'D' -> D
    | 'M' -> M
```

The compiler doesn't like that! What happens if we get some other character?

This is a great example of how exhaustive pattern matching can force you to think about missing requirements.

So, what should be done for bad input. How about printing an error message?

Let's try again and add a case to handle all other characters:

```
let charToRomanDigit =
    function
    | 'I' -> I
    | 'V' -> V
    | 'X' -> X
    | 'L' -> L
    | 'C' -> C
    | 'D' -> D
    | 'M' -> M
    | ch -> eprintf "%c is not a valid character" ch
```

The compiler doesn't like that either! The normal cases return a valid `RomanDigit` but the error case returns `unit`. As we saw in the earlier post, every branch *must* return the same type.

How can we fix this? We could throw an exception, but that seems a bit excessive. If we think about it some more, there's no way that `charToRomanDigit` can *always* return a valid `RomanDigit`. Sometimes it can, and sometimes it can't. In other words, we need to use something like an option type here.

But on further consideration, we might also want the caller to know what the bad char was. So we need to create our own little variant on the option type to hold both cases.

Here's the fixed up version:

```
type ParsedChar =
    | Digit of RomanDigit
    | BadChar of char

let charToRomanDigit =
    function
    | 'I' -> Digit I
    | 'V' -> Digit V
    | 'X' -> Digit X
    | 'L' -> Digit L
    | 'C' -> Digit C
    | 'D' -> Digit D
    | 'M' -> Digit M
    | ch -> BadChar ch
```

Note that I have removed the error message. Since the bad char is being returned, the caller can print its own message for the `BadChar` case.

Next, we should check the function signature to make sure it is what we expect:

```
charToRomanDigit : char -> ParsedChar
```

That looks good.

Now, how can we convert a string into these digits? We convert the string to a char array, convert that into a list, and then do a final conversion using `charToRomanDigit`.

```
let toRomanDigitList s =
    s.ToCharArray() // error FS0072
    |> List.ofArray
    |> List.map charToRomanDigit
```

But the compiler complains again with "FS0072: Lookup on object of indeterminate type",

That typically happens when you use a method rather than a function. Any object could implement `.ToCharArray()` so the type inference cannot tell what type is meant.

In this case, the solution is just to use an explicit type annotation on the parameter -- our first so far!

```
let toRomanDigitList (s:string) =
    s.ToCharArray()
    |> List.ofArray
    |> List.map charToRomanDigit
```

But look at the signature:

```
toRomanDigitList : string -> ParsedChar list
```

It still has the pesky `ParsedChar` in it rather than `RomanDigits`. How do we want to proceed? Answer, let's pass the buck again and let someone else deal with it!

"Passing the buck" in this case is actually a good design principle. This function doesn't know what its clients might want to do -- some might want to ignore errors, while others might want to fail fast. So just pass back the information and let them decide.

In this case, the client is the top level function that creates a `RomanNumeral` type. Here's our first attempt:

```
// convert a string to a RomanNumeral
let toRomanNumeral s =
    toRomanDigitList s
    |> RomanNumeral
```

The compiler is not happy -- the `RomanNumeral` constructor requires a list of `RomanDigits`, but the `toRomanDigitList` is giving us a list of `ParsedChars` instead.

Now we finally *do* have to commit to an error handling policy. Let's choose to ignore bad chars, but print out errors when they occur. We'll use the `List.choose` function for this. It's similar to `List.map`, but in addition has a filter built into it. Elements that are valid (`Some something`) are returned, but elements that are `None` are filtered out.

Our choose function should thus do the following:

- For valid digits return `Some digit`
- For the invalid `BadChars`, print the error message and return `None`.

If we do this, the output of `List.choose` will be a list of `RomanDigits`, exactly as needed as the input to the `RomanNumeral` constructor.

Here is everything put together:

```
/// Convert a string to a RomanNumeral
/// Does not validate the input.E.g. "IVIV" would be valid
let toRomanNumeral s =
    toRomanDigitList s
    |> List.choose (
        function
        | Digit digit ->
          Some digit
        | BadChar ch ->
          eprintfn "%c is not a valid character" ch
          None
        )
    |> RomanNumeral
```

Let's test!

```
// test good cases

"IIII"  |> toRomanNumeral
"IV"  |> toRomanNumeral
"VI"  |> toRomanNumeral
"IX"  |> toRomanNumeral
"MCMLXXIX"  |> toRomanNumeral
"MCMXLIV" |> toRomanNumeral
"" |> toRomanNumeral

// error cases
"MC?I" |> toRomanNumeral
"abc" |> toRomanNumeral
```

Ok, everything is good so far. Let's move on to validation.

Validation rules

The validation rules were not listed in the requirements, so let's put down our best guess based on what we know about Roman numerals:

- Five in a row of any digit is not allowed
- Some digits are allowed in runs of up to 4. They are I,X,C, and M. The others (V,L,D) can only appear singly.
- Some lower digits can come before a higher digit, but only if they appear singly. E.g. "IX" is ok but "IIIX" is not.
- But this is only for pairs of digits. Three ascending numbers in a row is invalid. E.g. "IX" is ok but "IXC" is not.
- A single digit with no runs is always allowed

We can convert these requirements into a pattern matching function as follows:

```
let runsAllowed =
    function
    | I | X | C | M -> true
    | V | L | D -> false

let noRunsAllowed  = runsAllowed >> not

// check for validity
let rec isValidDigitList digitList =
    match digitList with

    // empty list is valid
    | [] -> true

    // A run of 5 or more anything is invalid
    // Example:  XXXXX
    | d1::d2::d3::d4::d5::_
        when d1=d2 && d1=d3 && d1=d4 && d1=d5 ->
            false

    // 2 or more non-runnable digits is invalid
    // Example:  VV
    | d1::d2::_
        when d1=d2 && noRunsAllowed d1 ->
            false

    // runs of 2,3,4 in the middle are invalid if next digit is higher
    // Example:  IIIX
    | d1::d2::d3::d4::higher::ds
        when d1=d2 && d1=d3 && d1=d4
        && runsAllowed d1 // not really needed because of the order of matching
        && higher > d1 ->
            false

    | d1::d2::d3::higher::ds
        when d1=d2 && d1=d3
        && runsAllowed d1
        && higher > d1 ->
            false

    | d1::d2::higher::ds
        when d1=d2
        && runsAllowed d1
        && higher > d1 ->
            false

    // three ascending numbers in a row is invalid
    // Example:  IVX
    | d1::d2::d3::_  when d1<d2 && d2<= d3 ->
        false
```

```
    // A single digit with no runs is always allowed
    | _::ds ->
        // check the remainder of the list
        isValidDigitList ds
```

Again, note that "equality" and "less than" did not need to be defined.

And let's test the validation:

```
// test valid
let validList = [
    [I;I;I;I]
    [I;V]
    [I;X]
    [I;X;V]
    [V;X]
    [X;I;V]
    [X;I;X]
    [X;X;I;I]
    ]

let testValid = validList |> List.map isValidDigitList

let invalidList = [
    // Five in a row of any digit is not allowed
    [I;I;I;I;I]
    // Two in a row for V,L, D is not allowed
    [V;V]
    [L;L]
    [D;D]
    // runs of 2,3,4 in the middle are invalid if next digit is higher
    [I;I;V]
    [X;X;X;M]
    [C;C;C;C;D]
    // three ascending numbers in a row is invalid
    [I;V;X]
    [X;L;D]
    ]
let testInvalid = invalidList |> List.map isValidDigitList
```

Finally, we add a top level function to test validity of the `RomanNumeral` type itself.

```
// top level check for validity
let isValid (RomanNumeral digitList) =
    isValidDigitList digitList


// test good cases
"IIII"  |> toRomanNumeral |> isValid
"IV"  |> toRomanNumeral |> isValid
"" |> toRomanNumeral |> isValid

// error cases
"IIXX" |> toRomanNumeral |> isValid
"VV" |> toRomanNumeral |> isValid

// grand finale
[ "IIII"; "XIV"; "MMDXC";
"IIXX"; "VV"; ]
|> List.map toRomanNumeral
|> List.iter (function
    | n when isValid n ->
        printfn "%A is valid and its integer value is %i" n (toInt n)
    | n ->
        printfn "%A is not valid" n
    )
```

The entire code for the first version

Here's all the code in one module:

```
module RomanNumeralsV1 =

    // ==========================================
    // Types
    // ==========================================

    type RomanDigit = I | V | X | L | C | D | M
    type RomanNumeral = RomanNumeral of RomanDigit list

    // ==========================================
    // Output logic
    // ==========================================

    /// Converts a single RomanDigit to an integer
    let digitToInt =
        function
        | I -> 1
        | V -> 5
        | X -> 10
        | L -> 50
```

```
        | C -> 100
        | D -> 500
        | M -> 1000

    /// converts a list of digits to an integer
    let rec digitsToInt =
        function

        // empty is 0
        | [] -> 0

        // special case when a smaller comes before larger
        // convert both digits and add the difference to the sum
        // Example: "IV" and "CM"
        | smaller::larger::ns when smaller < larger ->
            (digitToInt larger - digitToInt smaller)  + digitsToInt ns

        // otherwise convert the digit and add to the sum
        | digit::ns ->
            digitToInt digit + digitsToInt ns

    /// converts a RomanNumeral to an integer
    let toInt (RomanNumeral digits) = digitsToInt digits

    // ==========================================
    // Input logic
    // ==========================================

    type ParsedChar =
        | Digit of RomanDigit
        | BadChar of char

    let charToRomanDigit =
        function
        | 'I' -> Digit I
        | 'V' -> Digit V
        | 'X' -> Digit X
        | 'L' -> Digit L
        | 'C' -> Digit C
        | 'D' -> Digit D
        | 'M' -> Digit M
        | ch -> BadChar ch

    let toRomanDigitList (s:string) =
        s.ToCharArray()
        |> List.ofArray
        |> List.map charToRomanDigit

    /// Convert a string to a RomanNumeral
    /// Does not validate the input.E.g. "IVIV" would be valid
    let toRomanNumeral s =
        toRomanDigitList s
        |> List.choose (
```

```
            function
            | Digit digit ->
                Some digit
            | BadChar ch ->
                eprintfn "%c is not a valid character" ch
                None
            )
        |> RomanNumeral

    // ==========================================
    // Validation logic
    // ==========================================

    let runsAllowed =
        function
        | I | X | C | M -> true
        | V | L | D -> false

    let noRunsAllowed  = runsAllowed >> not

    // check for validity
    let rec isValidDigitList digitList =
        match digitList with

        // empty list is valid
        | [] -> true

        // A run of 5 or more anything is invalid
        // Example:  XXXXX
        | d1::d2::d3::d4::d5::_
            when d1=d2 && d1=d3 && d1=d4 && d1=d5 ->
                false

        // 2 or more non-runnable digits is invalid
        // Example:  VV
        | d1::d2::_
            when d1=d2 && noRunsAllowed d1 ->
                false

        // runs of 2,3,4 in the middle are invalid if next digit is higher
        // Example:  IIIX
        | d1::d2::d3::d4::higher::ds
            when d1=d2 && d1=d3 && d1=d4
            && runsAllowed d1 // not really needed because of the order of matching
            && higher > d1 ->
                false

        | d1::d2::d3::higher::ds
            when d1=d2 && d1=d3
            && runsAllowed d1
            && higher > d1 ->
                false
```

```
        | d1::d2::higher::ds
            when d1=d2
            && runsAllowed d1
            && higher > d1 ->
                false

        // three ascending numbers in a row is invalid
        // Example:  IVX
        | d1::d2::d3::_  when d1<d2 && d2<= d3 ->
            false

        // A single digit with no runs is always allowed
        | _::ds ->
            // check the remainder of the list
            isValidDigitList ds

    // top level check for validity
    let isValid (RomanNumeral digitList) =
        isValidDigitList digitList
```

Second version

The code works, but there is something that's bugging me about it. The validation logic seems very complicated. Surely the Romans didn't have to think about all of this?

And also, I can think of examples that should fail validation, but pass, such as "VIV":

```
"VIV" |> toRomanNumeral |> isValid
```

We could try to tighten up our validation rules, but let's try another tack. Complicated logic is often a sign that you don't quite understand the domain properly.

In other words -- could we change the internal model to make everything simpler?

What about if we stopped trying to map letters to digits, and created a domain that mapped how the Romans thought it. In this model "I", "II", "III", "IV" and so on would each be a separate digit.

Let's run with it and see what happens.

Here's the new types for the domain. I now have a digit type for every possible digit. The `RomanNumeral` type stays the same.

```
type RomanDigit =
    | I | II | III | IIII
    | IV | V
    | IX | X | XX | XXX | XXXX
    | XL | L
    | XC | C | CC | CCC | CCCC
    | CD | D
    | CM | M | MM | MMM | MMMM
type RomanNumeral = RomanNumeral of RomanDigit list
```

Output: second version

Next, converting a single `RomanDigit` to an integer is the same as before, but with more cases:

```
/// Converts a single RomanDigit to an integer
let digitToInt =
    function
    | I -> 1 | II -> 2 | III -> 3 | IIII -> 4
    | IV -> 4 | V -> 5
    | IX -> 9 | X -> 10 | XX -> 20 | XXX -> 30 | XXXX -> 40
    | XL -> 40 | L -> 50
    | XC -> 90 | C -> 100 | CC -> 200 | CCC -> 300 | CCCC -> 400
    | CD -> 400 | D -> 500
    | CM -> 900 | M -> 1000 | MM -> 2000 | MMM -> 3000 | MMMM -> 4000

// tests
I  |> digitToInt
III  |> digitToInt
V  |> digitToInt
CM  |> digitToInt
```

Calculating the sum of the digits is now trivial. No special cases needed:

```
/// converts a list of digits to an integer
let digitsToInt list =
    list |> List.sumBy digitToInt

// tests
[IIII]  |> digitsToInt
[IV]  |> digitsToInt
[V;I]  |> digitsToInt
[IX]  |> digitsToInt
[M;CM;L;X;X;IX]  |> digitsToInt // 1979
[M;CM;XL;IV] |> digitsToInt // 1944
```

Finally, the top level function is identical:

```
/// converts a RomanNumeral to an integer
let toInt (RomanNumeral digits) = digitsToInt digits

// test
let x = RomanNumeral [M;CM;LX;X;IX]
x |> toInt
```

Input: second version

For the input parsing, we'll keep the `ParsedChar` type. But this time, we have to match 1,2,3, or 4 chars at a time. That means we can't just pull off one character like we did in the first version -- we have to match in the main loop. This means the loop now has to be recursive.

Also, we want to convert IIII into a single `IIII` digit rather than 4 separate `I` digits, so we put the longest matches at the front.

```
type ParsedChar =
    | Digit of RomanDigit
    | BadChar of char

let rec toRomanDigitListRec charList =
    match charList with
    // match the longest patterns first

    // 4 letter matches
    | 'I'::'I'::'I'::'I'::ns ->
        Digit IIII :: (toRomanDigitListRec ns)
    | 'X'::'X'::'X'::'X'::ns ->
        Digit XXXX :: (toRomanDigitListRec ns)
    | 'C'::'C'::'C'::'C'::ns ->
        Digit CCCC :: (toRomanDigitListRec ns)
    | 'M'::'M'::'M'::'M'::ns ->
        Digit MMMM :: (toRomanDigitListRec ns)

    // 3 letter matches
    | 'I'::'I'::'I'::ns ->
        Digit III :: (toRomanDigitListRec ns)
    | 'X'::'X'::'X'::ns ->
        Digit XXX :: (toRomanDigitListRec ns)
    | 'C'::'C'::'C'::ns ->
        Digit CCC :: (toRomanDigitListRec ns)
    | 'M'::'M'::'M'::ns ->
        Digit MMM :: (toRomanDigitListRec ns)

    // 2 letter matches
    | 'I'::'I'::ns ->
        Digit II :: (toRomanDigitListRec ns)
    | 'X'::'X'::ns ->
        Digit XX :: (toRomanDigitListRec ns)
```

```
    | 'C'::'C'::ns ->
        Digit CC :: (toRomanDigitListRec ns)
    | 'M'::'M'::ns ->
        Digit MM :: (toRomanDigitListRec ns)

    | 'I'::'V'::ns ->
        Digit IV :: (toRomanDigitListRec ns)
    | 'I'::'X'::ns ->
        Digit IX :: (toRomanDigitListRec ns)
    | 'X'::'L'::ns ->
        Digit XL :: (toRomanDigitListRec ns)
    | 'X'::'C'::ns ->
        Digit XC :: (toRomanDigitListRec ns)
    | 'C'::'D'::ns ->
        Digit CD :: (toRomanDigitListRec ns)
    | 'C'::'M'::ns ->
        Digit CM :: (toRomanDigitListRec ns)

    // 1 letter matches
    | 'I'::ns ->
        Digit I :: (toRomanDigitListRec ns)
    | 'V'::ns ->
        Digit V :: (toRomanDigitListRec ns)
    | 'X'::ns ->
        Digit X :: (toRomanDigitListRec ns)
    | 'L'::ns ->
        Digit L :: (toRomanDigitListRec ns)
    | 'C'::ns ->
        Digit C :: (toRomanDigitListRec ns)
    | 'D'::ns ->
        Digit D :: (toRomanDigitListRec ns)
    | 'M'::ns ->
        Digit M :: (toRomanDigitListRec ns)

    // bad letter matches
    | badChar::ns ->
        BadChar badChar :: (toRomanDigitListRec ns)

    // 0 letter matches
    | [] ->
        []
```

Well, this is much longer than the first version, but otherwise basically the same.

The top level functions are unchanged.

```
let toRomanDigitList (s:string) =
    s.ToCharArray()
    |> List.ofArray
    |> toRomanDigitListRec

/// Convert a string to a RomanNumeral
let toRomanNumeral s =
    toRomanDigitList s
    |> List.choose (
        function
        | Digit digit ->
            Some digit
        | BadChar ch ->
            eprintfn "%c is not a valid character" ch
            None
        )
    |> RomanNumeral

// test good cases
"IIII"  |> toRomanNumeral
"IV"  |> toRomanNumeral
"VI"  |> toRomanNumeral
"IX"  |> toRomanNumeral
"MCMLXXIX"  |> toRomanNumeral
"MCMXLIV" |> toRomanNumeral
"" |> toRomanNumeral

// error cases
"MC?I" |> toRomanNumeral
"abc" |> toRomanNumeral
```

Validation: second version

Finally, let's see how the new domain model affects the validation rules. Now, the rules are *much* simpler. In fact, there is only one.

- Each digit must be smaller than the preceding digit

```
// check for validity
let rec isValidDigitList digitList =
    match digitList with

    // empty list is valid
    | [] -> true

    // a following digit that is equal or larger is an error
    | d1::d2::_
        when d1 <= d2  ->
            false

    // A single digit is always allowed
    | _::ds ->
        // check the remainder of the list
        isValidDigitList ds

// top level check for validity
let isValid (RomanNumeral digitList) =
    isValidDigitList digitList

// test good cases
"IIII"  |> toRomanNumeral |> isValid
"IV"  |> toRomanNumeral |> isValid
"" |> toRomanNumeral |> isValid

// error cases
"IIXX" |> toRomanNumeral |> isValid
"VV" |> toRomanNumeral |> isValid
```

Alas, after all that, we still didn't fix the bad case that triggered the rewrite!

```
"VIV" |> toRomanNumeral |> isValid
```

There is a not-too-complicated fix for this, but I think it's time to leave it alone now!

The entire code for the second version

Here's all the code in one module for the second version:

```
module RomanNumeralsV2 =

    // ==========================================
    // Types
    // ==========================================

    type RomanDigit =
```

```
    | I | II | III | IIII
    | IV | V
    | IX | X | XX | XXX | XXXX
    | XL | L
    | XC | C | CC | CCC | CCCC
    | CD | D
    | CM | M | MM | MMM | MMMM
type RomanNumeral = RomanNumeral of RomanDigit list

// ==========================================
// Output logic
// ==========================================

/// Converts a single RomanDigit to an integer
let digitToInt =
    function
    | I -> 1 | II -> 2 | III -> 3 | IIII -> 4
    | IV -> 4 | V -> 5
    | IX -> 9 | X -> 10 | XX -> 20 | XXX -> 30 | XXXX -> 40
    | XL -> 40 | L -> 50
    | XC -> 90 | C -> 100 | CC -> 200 | CCC -> 300 | CCCC -> 400
    | CD -> 400 | D -> 500
    | CM -> 900 | M -> 1000 | MM -> 2000 | MMM -> 3000 | MMMM -> 4000

/// converts a RomanNumeral to an integer
let toInt (RomanNumeral digits) = digitsToInt digits

// ==========================================
// Input logic
// ==========================================

type ParsedChar =
    | Digit of RomanDigit
    | BadChar of char

let rec toRomanDigitListRec charList =
    match charList with
    // match the longest patterns first

    // 4 letter matches
    | 'I'::'I'::'I'::'I'::ns ->
        Digit IIII :: (toRomanDigitListRec ns)
    | 'X'::'X'::'X'::'X'::ns ->
        Digit XXXX :: (toRomanDigitListRec ns)
    | 'C'::'C'::'C'::'C'::ns ->
        Digit CCCC :: (toRomanDigitListRec ns)
    | 'M'::'M'::'M'::'M'::ns ->
        Digit MMMM :: (toRomanDigitListRec ns)

    // 3 letter matches
    | 'I'::'I'::'I'::ns ->
        Digit III :: (toRomanDigitListRec ns)
    | 'X'::'X'::'X'::ns ->
```

```
        Digit XXX :: (toRomanDigitListRec ns)
    | 'C'::'C'::'C'::ns ->
        Digit CCC :: (toRomanDigitListRec ns)
    | 'M'::'M'::'M'::ns ->
        Digit MMM :: (toRomanDigitListRec ns)

    // 2 letter matches
    | 'I'::'I'::ns ->
        Digit II :: (toRomanDigitListRec ns)
    | 'X'::'X'::ns ->
        Digit XX :: (toRomanDigitListRec ns)
    | 'C'::'C'::ns ->
        Digit CC :: (toRomanDigitListRec ns)
    | 'M'::'M'::ns ->
        Digit MM :: (toRomanDigitListRec ns)

    | 'I'::'V'::ns ->
        Digit IV :: (toRomanDigitListRec ns)
    | 'I'::'X'::ns ->
        Digit IX :: (toRomanDigitListRec ns)
    | 'X'::'L'::ns ->
        Digit XL :: (toRomanDigitListRec ns)
    | 'X'::'C'::ns ->
        Digit XC :: (toRomanDigitListRec ns)
    | 'C'::'D'::ns ->
        Digit CD :: (toRomanDigitListRec ns)
    | 'C'::'M'::ns ->
        Digit CM :: (toRomanDigitListRec ns)

    // 1 letter matches
    | 'I'::ns ->
        Digit I :: (toRomanDigitListRec ns)
    | 'V'::ns ->
        Digit V :: (toRomanDigitListRec ns)
    | 'X'::ns ->
        Digit X :: (toRomanDigitListRec ns)
    | 'L'::ns ->
        Digit L :: (toRomanDigitListRec ns)
    | 'C'::ns ->
        Digit C :: (toRomanDigitListRec ns)
    | 'D'::ns ->
        Digit D :: (toRomanDigitListRec ns)
    | 'M'::ns ->
        Digit M :: (toRomanDigitListRec ns)

    // bad letter matches
    | badChar::ns ->
        BadChar badChar :: (toRomanDigitListRec ns)

    // 0 letter matches
    | [] ->
        []
```

```
    let toRomanDigitList (s:string) =
        s.ToCharArray()
        |> List.ofArray
        |> toRomanDigitListRec

    /// Convert a string to a RomanNumeral
    /// Does not validate the input.E.g. "IVIV" would be valid
    let toRomanNumeral s =
        toRomanDigitList s
        |> List.choose (
            function
            | Digit digit ->
                Some digit
            | BadChar ch ->
                eprintfn "%c is not a valid character" ch
                None
            )
        |> RomanNumeral

    // ==========================================
    // Validation logic
    // ==========================================

    // check for validity
    let rec isValidDigitList digitList =
        match digitList with

        // empty list is valid
        | [] -> true

        // a following digit that is equal or larger is an error
        | d1::d2::_
            when d1 <= d2  ->
                false

        // A single digit is always allowed
        | _::ds ->
            // check the remainder of the list
            isValidDigitList ds

    // top level check for validity
    let isValid (RomanNumeral digitList) =
        isValidDigitList digitList
```

Comparing the two versions

Which version did you like better? The second one is more longwinded because it has many more cases, but on the other hand, the actual logic is the same or simpler in all areas, with no special cases. And as a result, the total number of lines of code is about the same for

both versions.

Overall, I prefer the second implementation because of the lack of special cases.

As a fun experiment, try writing the same code in C# or your favorite imperative language!

Making it object-oriented

Finally, let's see how we might make this object oriented. We don't care about the helper functions, so we probably just need three methods:

- A static constructor
- A method to convert to a int
- A method to convert to a string

And here they are:

```
type RomanNumeral with

    static member FromString s =
        toRomanNumeral s

    member this.ToInt() =
        toInt this

    override this.ToString() =
        sprintf "%A" this
```

Note: you can ignore the compiler warning about deprecated overrides.

Let's use this in an object oriented way now:

```
let r = RomanNumeral.FromString "XXIV"
let s = r.ToString()
let i = r.ToInt()
```

Summary

In this post we've seen lots and lots of pattern matching!

But again, as with the last post, what's equally important is that we've seen how easy it is to create a properly designed internal model for very trivial domains. And again, our internal model used no primitive types -- there is no excuse not to create lots of little types in order to

represent the domain better. For example, the `ParsedChar` type -- would you have bothered to create that in C#?

And as should be clear, the choice of an internal model can make a lot of difference to the complexity of the design. But if and when we do refactor, the compiler will almost always warn us if we have forgotten something.

F# is not just about functions; the powerful type system is another key ingredient. And just as with functions, understanding the type system is critical to being fluent and comfortable in the language.

In addition to the common .NET types. F# has some other types that are very common in functional languages but not available in imperative languages like C# or Java.

This series introduces these types and how to use them.

- Understanding F# types: Introduction. A new world of types.
- Overview of types in F#. A look at the big picture.
- Type abbreviations. Also known as aliases.
- Tuples. Multiplying types together.
- Records. Extending tuples with labels.
- Discriminated Unions. Adding types together.
- The Option type. And why it is not null or nullable.
- Enum types. Not the same as a union type.
- Built-in .NET types. Ints, strings, bools, etc.
- Units of measure. Type safety for numerics.
- Understanding type inference. Behind the magic curtain.

Understanding F# types: Introduction

NOTE: Before reading this series, I suggest that you read the "thinking functionally" and "expressions and syntax" series as a prerequisite.

F# is not just about functions; the powerful type system is another key ingredient. And just as with functions, understanding the type system is critical to being fluent and comfortable in the language.

Now, so far we have seen some basic types that can be used as input and output to functions:

- Primitive types such as `int` , `float` , `string` , and `bool`
- Simple function types such as `int->int`
- The `unit` type
- Generic types.

None of these types should be unfamiliar. Analogues of these are available in C# and other imperative languages.

But in this series we are going to introduce some new kinds of types that are very common in functional languages but uncommon in imperative languages.

The extended types we will look at in this series are:

- Tuples
- Records
- Unions
- The Option type
- Enum types

For all these types, we will discuss both the abstract principles and the details of how to use them in practice.

Lists and other recursive data types are also very important types, but there is so much to say about them that they will need their own series!

Overview of types in F#

Before we dive into all the specific types, let's look at the big picture.

What are types for?

If you are coming from an object-oriented design background, one of the paradigm shifts involved in "thinking functionally" is to change how you think about types.

A well designed object-oriented program will have a strong focus on behavior rather than data, so it will use a lot of polymorphism, either using "duck-typing" or explicit interfaces, and will try to avoid having explicit knowledge of the actual concrete classes being passed around.

A well designed functional program, on the other hand, will have a strong focus on *data types* rather than behavior. F# puts much more emphasis on designing types correctly than an imperative language such as C#, and many of the examples in this series and later series will focus on creating and refining type definitions.

So what is a type? Types are surprisingly hard to define. One definition from a well known textbook says:

> "A type system is a tractable syntactic method of proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute"
> *(Benjamin Pierce, Types and Programming Languages)*

Ok, that definition is a bit technical. So let's turn it around -- what do we use types for in practice? In the context of F#, you can think of types as being used in two main ways:

- Firstly, as an *annotation to a value* that allows certain checks to be made, especially at compile time. In other words, types allow you to have "compile time unit tests".
- Second, as *domains* for functions to act upon. That is, a type is a sort of data modeling tool that allows you to represent a real world domain in your code.

These two definitions interact. The better the type definitions reflect the real-world domain, the better they will statically encode the business rules. And the better they statically encode the business rules, the better the "compile time unit tests" work. In the ideal scenario, if your program compiles, then it really is correct!

What kinds of types are there?

F# is a hybrid language, so it has a mixture of types: some from its functional background, and some from its object-oriented background.

Generally, the types in F# can be grouped into the following categories:

- **Common .NET types**. These are types that conform to the .NET Common Language Infrastructure (CLI), and which are easily portable to every .NET language.
- **F# specific types**. These are types that are part of the F# language and are designed for pure functional programming.

If you are familiar with C#, you will know all the CLI types. They include:

- Built-in value types (int, bool, etc).
- Built-in reference types (string, etc).
- User-defined value types (enum and struct).
- Classes and interfaces
- Delegates
- Arrays

The F# specific types include:

- Function types (not the same as delegates or C# lambdas)
- The unit type
- Tuples (now part of .NET 4.0)
- Records
- Discriminated Unions
- Option types
- Lists (not the same as the .NET List class)

I strongly recommend that when creating new types you stick with the F# specific types rather than using classes. They have a number of advantages over the CLI types, such as:

- They are immutable
- They cannot be null
- They have built-in structural equality and comparison
- They have built-in pretty printing

Sum and Product types

The key to understanding the power of types in F# is that most new types are constructed by from other types using two basic operations: **sum** and **product**.

That is, in F# you can define new types almost as if you were doing algebra:

```
define typeZ = typeX "plus" typeY
define typeW = typeX "times" typeZ
```

I will hold off explaining what **sum** and **product** mean in practice until we get to the detailed discussion of tuples (products) and discriminated union (sum) types later in this series.

The key point is that an infinite number of new types can be made by combining existing types together using these "product" and "sum" methods in various ways. Collectively these are called "algebraic data types" or ADTs (not to be confused with *abstract data types*, also called ADTs). Algebraic data types can be used to model anything, including lists, trees, and other recursive types.

The sum or "union" types, in particular, are very valuable, and once you get used to them, you will find them indispensible!

How types are defined

Every type definition is similar, even though the specific details may vary. All type definitions start with a " `type` " keyword, followed by an identifier for the type, followed by any generic type parameters, followed by the definition. For example, here are some type definitions for a variety of types:

```
type A = int * int
type B = {FirstName:string; LastName:string}
type C = Circle of int | Rectangle of int * int
type D = Day | Month | Year
type E<'a> = Choice1 of 'a | Choice2 of 'a * 'a

type MyClass(initX:int) =
   let x = initX
   member this.Method() = printf "x=%i" x
```

As we said in a previous post, there is a special syntax for defining new types that is different from the normal expression syntax. So do be aware of this difference.

Types can *only* be declared in namespaces or modules. But that doesn't mean you always have to create them at the top level -- you can create types in nested modules if you need to hide them.

```
module sub =
    // type declared in a module
    type A = int * int

    module private helper =
        // type declared in a submodule
        type B = B of string list

        //internal access is allowed
        let b = B ["a";"b"]

//outside access not allowed
let b = sub.helper.B ["a";"b"]
```

Types *cannot* be declared inside functions.

```
let f x =
    type A = int * int  //unexpected keyword "type"
    x * x
```

Constructing and deconstructing types

After a type is defined, instances of the type are created using a "constructor" expression that often looks quite similar to the type definition itself.

```
let a = (1,1)
let b = { FirstName="Bob"; LastName="Smith" }
let c = Circle 99
let c' = Rectangle (2,1)
let d = Month
let e = Choice1 "a"
let myVal = MyClass 99
myVal.Method()
```

What is interesting is that the *same* "constructor" syntax is also used to "deconstruct" the type when doing pattern matching:

```
let a = (1,1)                                  // "construct"
let (a1,a2) = a                                // "deconstruct"

let b = { FirstName="Bob"; LastName="Smith" }  // "construct"
let { FirstName = b1 } = b                     // "deconstruct"

let c = Circle 99                              // "construct"
match c with
| Circle c1 -> printf "circle of radius %i" c1 // "deconstruct"
| Rectangle (c2,c3) -> printf "%i %i" c2 c3    // "deconstruct"

let c' = Rectangle (2,1)                       // "construct"
match c' with
| Circle c1 -> printf "circle of radius %i" c1 // "deconstruct"
| Rectangle (c2,c3) -> printf "%i %i" c2 c3    // "deconstruct"
```

As you read through this series, pay attention to how the constructors are used in both ways.

Field guide to the "type" keyword

The same "type" keyword is used to define all the F# types, so they can all look very similar if you are new to F#. Here is a quick list of these types and how to tell the difference between them.

Type	Example	Distinguishing features
Abbrev (Alias)	`type ProductCode = string` `type transform<'a> = 'a -> 'a`	Uses equal sign only.

Tuple	`//not explicitly defined with type keyword`
	`//usage`
	`let t = 1,2`
	`let s = (3,4)`
		Always available to be used and are not explicitly defined with the `type` keyword. Usage indicated by comma (with optional parentheses).

<table>
<tr><td>Record</td><td>type Product =
{code:ProductCode;
price:float }
type Message<'a> = {id:int;
body:'a}

//usage
let p = {code="X123";
price=9.99}
let m = {id=1;
body="hello"}</td><td>Curly braces.
Uses semicolon to separate fields.</td></tr>
<tr><td>Discriminated Union</td><td>type MeasurementUnit = Cm |
Inch | Mile
type Name =
    | Nickname of string
    | FirstLast of string *
string
type Tree<'a> =
    | E
    | T of Tree<'a> * 'a *
Tree<'a>
//usage
let u = Inch
let name = Nickname("John")
let t = T(E,"John",E)</td><td>Vertical bar character.
Uses "of" for types.</td></tr>
<tr><td>Enum</td><td>type Gender = | Male = 1 |
Female = 2
//usage
let g = Gender.Male</td><td>Similar to Unions, but uses equals and an int value</td></tr>
<tr><td></td><td></td><td>Has function-style parameter list after name for use as</td></tr>
</table>

<table>
<tr>
<td>Class</td>
<td>

```
type Product (code:string, price:float) =
   let isFree = price=0.0
   new (code) = Product(code,0.0)
   member this.Code = code
   member this.IsFree = isFree

//usage
let p = Product("X123",9.99)
let p2 = Product("X123")
```

</td>
<td>Has "member" keyword.
Has "new" keyword for secondary constructors.</td>
</tr>
<tr>
<td>Interface</td>
<td>

```
type IPrintable =
   abstract member Print : unit -> unit
```

</td>
<td>Same as class but all members are abstract.
Abstract members have colon and type signature rather than a concrete implementation.</td>
</tr>
<tr>
<td>Struct</td>
<td>

```
type Product=
   struct
      val code:string
      val price:float
      new(code) = { code = code; price = 0.0 }
   end

//usage
let p = Product()
let p2 = Product("X123")
```

</td>
<td>Has "struct" keyword.
Uses "val" to define fields.
Can have constructor.</td>
</tr>
</table>

Type abbreviations

Let's start with the simplest type definition, a type abbreviation or alias.

It has the form:

```
type [typename] = [existingType]
```

where "existing type" can be any type: one of the basic types we have already seen, or one of the extended types we will be seeing soon.

Some examples:

```
type RealNumber = float
type ComplexNumber = float * float
type ProductCode = string
type CustomerId = int
type AdditionFunction = int->int->int
type ComplexAdditionFunction =
       ComplexNumber-> ComplexNumber -> ComplexNumber
```

And so on -- pretty straightforward.

Type abbreviations are useful to provide documentation and avoid writing a signature repeatedly. In the above examples, `ComplexNumber` and `AdditionFunction` demonstrate this.

Another use is to provide some degree of decoupling between the usage of a type and the actual implementation of a type. In the above examples, `ProductCode` and `CustomerId` demonstrate this. I could easily change `CustomerId` to be a string without changing (most of) my code.

However, one thing is to note is that this really is just an alias or abbreviation; you are not actually creating a new type. So if I define a function that I explicitly say is an `AdditionFunction`:

```
type AdditionFunction = int->int->int
let f:AdditionFunction = fun a b -> a + b
```

the compiler will erase the alias and return a plain `int->int->int` as the function signature.

In particular, there is no true encapsulation. I could use an explicit `int` anywhere I used a `CustomerId` and the compiler would not complain. And if I had attempted to create safe versions of entity ids such as this:

```
type CustomerId = int
type OrderId = int
```

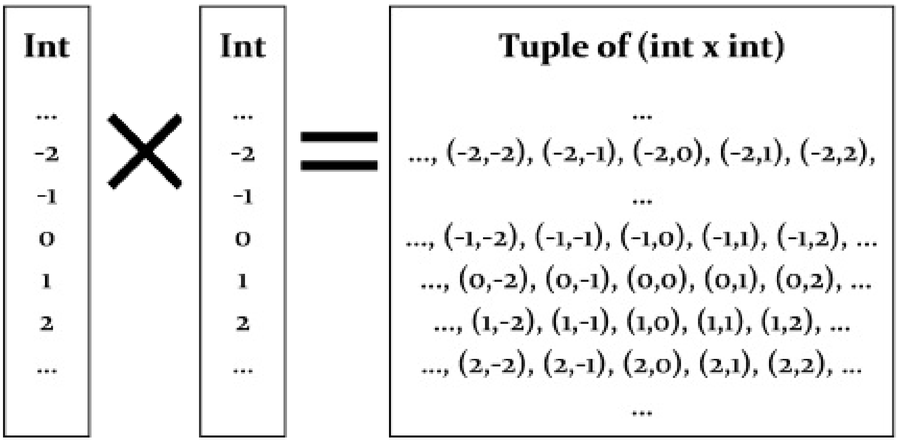
then I would be disappointed. There would be nothing preventing me from using an `OrderId` in place of a `CustomerId` and vice versa. To get true encapsulated types like this, we will need to use single case union types, as described in a later post.

Tuples

We're ready for our first extended type -- the tuple.

Let's start by stepping back again and looking at a type such as "int". As we hinted at before, rather than thinking of "int" as a abstract thing, you can think of it as concrete collection of all its possible values, namely the set {...,-3, -2, -1, 0, 2, 3, ...}.

So next, imagine two copies of this "int" collection. We can "multiply" them together by taking the Cartesian product of them; that is, making a new list of objects by picking every possible combination of the two "int" lists, as shown below:



As we have already seen, these pairs are called tuples in F#. And now you can see why they have the type signature that they do. In this example, the "int times int" type is called "`int * int`", and the star symbol means "multiply" of course! The valid instances of this new type are all the pairs: (-2,2),(-1,0), (2,2) and so on.

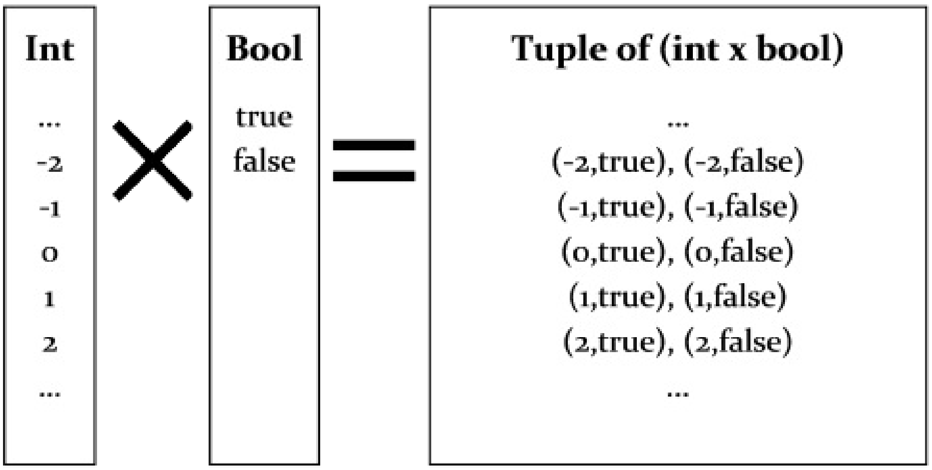
Let's see how they might be used in practice:

```
let t1 = (2,3)
let t2 = (-2,7)
```

Now if you evaluate the code above you will see that the types of t1 and t2 are `int*int` as expected.

```
val t1 : int * int = (2, 3)
val t2 : int * int = (-2, 7)
```

This "product" approach can be used to make tuples out of any mixture of types. Here is one for "int times bool".

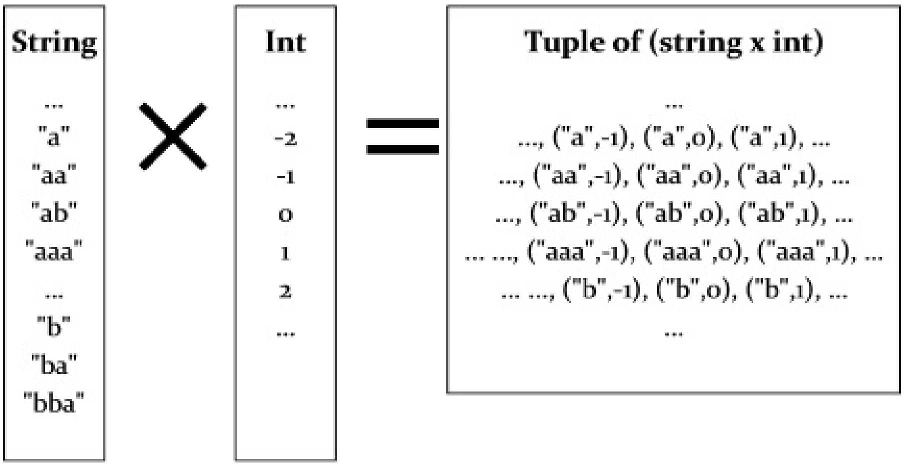


And here is the usage in F#. The tuple type above has the signature " `int*bool` ".

```
let t3 = (2,true)
let t4 = (7,false)

// the signatures are:
val t3 : int * bool = (2, true)
val t4 : int * bool = (7, false)
```

Strings can be used as well, of course. The universe of all possible strings is very large, but conceptually it is the same thing. The tuple type below has the signature " `string*int` ".

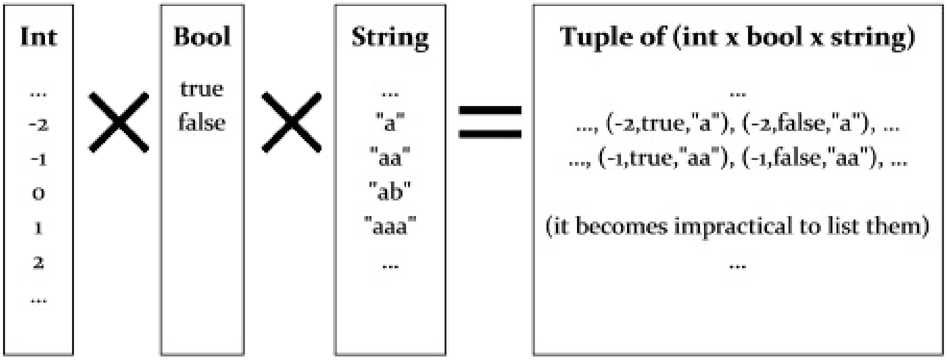


Test the usage and signatures:

```
let t5 = ("hello",42)
let t6 = ("goodbye",99)

// the signatures are:
val t5 : string * int = ("hello", 42)
val t6 : string * int = ("goodbye", 99)
```

And there is no reason to stop at multiplying just two types together. Why not three? Or four? For example, here is the type `int * bool * string`.



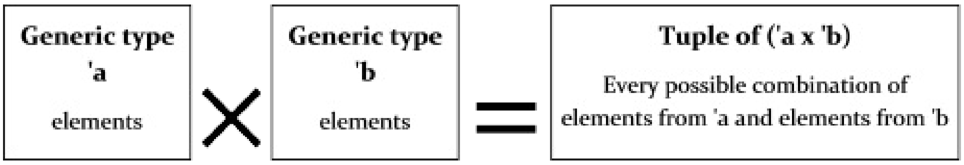
Test the usage and signatures:

```
let t7 = (42,true,"hello")

// the signature is:
val t7 : int * bool * string = (42, true, "hello")
```

Generic tuples

Generics can be used in tuples too.



The usage is normally associated with functions:

```
let genericTupleFn aTuple =
   let (x,y) = aTuple
   printfn "x is %A and y is %A" x y
```

And the function signature is:

```
val genericTupleFn : 'a * 'b -> unit
```

which means that " `genericTupleFn` " takes a generic tuple `('a * 'b)` and returns a `unit`

Tuples of complex types

Any kind of type can be used in a tuple: other tuples, classes, function types, etc. Here are some examples:

```
// define some types
type Person = {First:string; Last:string}
type Complex = float * float
type ComplexComparisonFunction = Complex -> Complex -> int

// define some tuples using them
type PersonAndBirthday = Person * System.DateTime
type ComplexPair = Complex * Complex
type ComplexListAndSortFunction = Complex list * ComplexComparisonFunction
type PairOfIntFunctions = (int->int) * (int->int)
```

Key points about tuples

Some key things to know about tuples are:

- A particular instance of a tuple type is a *single object*, similar to a two-element array in C#, say. When using them with functions they count as a *single* parameter.
- Tuple types cannot be given explicit names. The "name" of the tuple type is determined by the combination of types that are multiplied together.
- The order of the multiplication is important. So `int*string` is not the same tuple type as `string*int` .
- The comma is the critical symbol that defines tuples, not the parentheses. You can define tuples without the parentheses, although it can sometimes be confusing. In F#, if you see a comma, it is probably part of a tuple.

These points are very important -- if you don't understand them you will get confused quite quickly!

And it is worth re-iterating the point made in previous posts: *don't mistake tuples for multiple parameters in a function*.

```
// a function that takes a single tuple parameter
// but looks like it takes two ints
let addConfusingTuple (x,y) = x + y
```

Making and matching tuples

The tuple types in F# are somewhat more primitive than the other extended types. As you have seen, you don't need to explicitly define them, and they have no name.

It is easy to make a tuple -- just use a comma!

```
let x = (1,2)
let y = 1,2        // it's the comma you need, not the parentheses!
let z = 1,true,"hello",3.14   // create arbitrary tuples as needed
```

And as we have seen, to "deconstruct" a tuple, use the same syntax:

```
let z = 1,true,"hello",3.14   // "construct"
let z1,z2,z3,z4 = z           // "deconstruct"
```

When pattern matching like this, you must have the same number of elements, otherwise you will get an error:

```
let z1,z2 = z     // error FS0001: Type mismatch.
                  // The tuples have differing lengths
```

If you don't need some of the values, you can use the "don't care" symbol (the underscore) as a placeholder.

```
let _,z5,_,z6 = z     // ignore 1st and 3rd elements
```

As you might guess, a two element tuple is commonly called a "pair" and a three element tuple is called a "triple" and so on. In the special case of pairs, there are functions `fst` and `snd` which extract the first and second element.

```
let x = 1,2
fst x
snd x
```

They only work on pairs. Trying to use `fst` on a triple will give an error.

```
let x = 1,2,3
fst x             // error FS0001: Type mismatch.
                  // The tuples have differing lengths of 2 and 3
```

Using tuples in practice

Tuples have a number of advantages over other more complex types. They can be used on the fly because they are always available without being defined, and thus are perfect for small, temporary, lightweight structures.

Using tuples for returning multiple values

It is a common scenario that you want to return two values from a function rather than just one. For example, in the `TryParse` style functions, you want to return (a) whether the value was parsed and (b) if parsed, what the parsed value was.

Here is an implementation of `TryParse` for integers (assuming it did not already exist, of course):

```
let tryParse intStr =
   try
      let i = System.Int32.Parse intStr
      (true,i)
   with _ -> (false,0)  // any exception

//test it
tryParse "99"
tryParse "abc"
```

Here's another simple example that returns a pair of numbers:

```
// return word count and letter count in a tuple
let wordAndLetterCount (s:string) =
   let words = s.Split [|' '|]
   let letterCount = words |> Array.sumBy (fun word -> word.Length )
   (words.Length, letterCount)

//test
wordAndLetterCount "to be or not to be"
```

Creating tuples from other tuples

As with most F# values, tuples are immutable and the elements within them cannot be assigned to. So how do you change a tuple? The short answer is that you can't -- you must always create a new one.

Say that you need to write a function that, given a tuple, adds one to each element. Here's an obvious implementation:

```
let addOneToTuple aTuple =
   let (x,y,z) = aTuple
   (x+1,y+1,z+1)   // create a new one

// try it
addOneToTuple (1,2,3)
```

This seems a bit long winded -- is there a more compact way? Yes, because you can deconstruct a tuple directly in the parameters of a function, so that the function becomes a one liner:

```
let addOneToTuple (x,y,z) = (x+1,y+1,z+1)

// try it
addOneToTuple (1,2,3)
```

Equality

Tuples have an automatically defined equality operation: two tuples are equal if they have the same length and the values in each slot are equal.

```
(1,2) = (1,2)                      // true
(1,2,3,"hello") = (1,2,3,"bye")    // false
(1,(2,3),4) = (1,(2,3),4)          // true
```

Trying to compare tuples of different lengths is a type error:

```
(1,2) = (1,2,3)                    // error FS0001: Type mismatch
```

And the types in each slot must be the same as well:

```
(1,2,3) = (1,2,"hello")   // element 3 was expected to have type
                          // int but here has type string
(1,(2,3),4) = (1,2,(3,4)) // elements 2 & 3 have different types
```

Tuples also have an automatically defined hash value based on the values in the tuple, so that tuples can be used as dictionary keys without problems.

```
(1,2,3).GetHashCode()
```

Tuple representation

And as noted in a previous post, tuples have a nice default string representation, and can be serialized easily.

```
(1,2,3).ToString()
```

Records

As we noted in the previous post, plain tuples are useful in many cases. But they have some disadvantages too. Because all tuple types are pre-defined, you can't distinguish between a pair of floats used for geographic coordinates say, vs. a similar tuple used for complex numbers. And when tuples have more than a few elements, it is easy to get confused about which element is in which place.

In these situations, what you would like to do is *label* each slot in the tuple, which will both document what each element is for and force a distinction between tuples made from the same types.

Enter the "record" type. A record type is exactly that, a tuple where each element is labeled.

```
type ComplexNumber = { real: float; imaginary: float }
type GeoCoord = { lat: float; long: float }
```

A record type has the standard preamble: `type [typename] =` followed by curly braces. Inside the curly braces is a list of `label: type` pairs, separated by semicolons (remember, all lists in F# use semicolon separators -- commas are for tuples).

Let's compare the "type syntax" for a record type with a tuple type:

```
type ComplexNumberRecord = { real: float; imaginary: float }
type ComplexNumberTuple = float * float
```

In the record type, there is no "multiplication", just a list of labeled types.

> Relational database theory uses a similar "record type" concept. In the relational model, a relation is a (possibly empty) finite set of tuples all having the same finite set of attributes. This set of attributes is familiarly referred to as the set of column names.

Making and matching records

To create a record value, use a similar format to the type definition, but using equals signs after the labels. This is called a "record expression."

```
type ComplexNumberRecord = { real: float; imaginary: float }
let myComplexNumber = { real = 1.1; imaginary = 2.2 } // use equals!

type GeoCoord = { lat: float; long: float } // use colon in type
let myGeoCoord = { lat = 1.1; long = 2.2 }  // use equals in let
```

And to "deconstruct" a record, use the same syntax:

```
let myGeoCoord = { lat = 1.1; long = 2.2 }   // "construct"
let { lat=myLat; long=myLong } = myGeoCoord  // "deconstruct"
```

As always, if you don't need some of the values, you can use the underscore as a placeholder; or more cleanly, just leave off the unwanted label altogether.

```
let { lat=_; long=myLong2 } = myGeoCoord  // "deconstruct"
let { long=myLong3 } = myGeoCoord         // "deconstruct"
```

If you just need a single property, you can use dot notation rather than pattern matching.

```
let x = myGeoCoord.lat
let y = myGeoCoord.long
```

Note that you can leave a label off when deconstructing, but not when constructing:

```
let myGeoCoord = { lat = 1.1; }  // error FS0764: No assignment
                                 // given for field 'long'
```

One of the most noticeable features of record types is use of curly braces. Unlike C-style languages, curly braces are rarely used in F# -- only for records, sequences, computation expressions (of which sequences are a special case), and object expressions (creating implementations of interfaces on the fly). These other uses will be discussed later.

Label order

Unlike tuples, the order of the labels is not important. So the following two values are the same:

```
let myGeoCoordA = { lat = 1.1; long = 2.2 }
let myGeoCoordB = { long = 2.2; lat = 1.1 }   // same as above
```

Naming conflicts

In the examples above, we could construct a record by just using the label names " `lat` " and " `long` ". Magically, the compiler knew what record type to create. (Well, in truth, it was not really that magical, as only one record type had those exact labels.)

But what happens if there are two record types with the same labels? How can the compiler know which one you mean? The answer is that it can't -- it will use the most recently defined type, and in some cases, issue a warning. Try evaluating the following:

```
type Person1 = {first:string; last:string}
type Person2 = {first:string; last:string}
let p = {first="Alice"; last="Jones"}
```

What type is `p` ? Answer: `Person2` , which was the last type defined with those labels.

And if you try to deconstruct, you will get a warning about ambiguous field labels.

```
let {first=f; last=l} = p
```

How can you fix this? Simply by adding the type name as a qualifier to at least one of the labels.

```
let p = {Person1.first="Alice"; last="Jones"}
let { Person1.first=f; last=l} = p
```

If needed, you can even add a fully qualified name (with namespace). Here's an example using modules.

```
module Module1 =
  type Person = {first:string; last:string}

module Module2 =
  type Person = {first:string; last:string}

module Module3 =
  let p = {Module1.Person.first="Alice";
           Module1.Person.last="Jones"}
```

Of course, if you can ensure there is only one version in the local namespace, you can avoid having to do this at all.

```
module Module3b =
  open Module1                   // bring into the local namespace
  let p = {first="Alice"; last="Jones"}  // will be Module1.Person
```

The moral of the story is that when defining record types, you should try to use unique labels if possible, otherwise you will get ugly code at best, and unexpected behavior at worst.

> Note that in F#, unlike some other functional languages, two types with exactly the same structural definition are not the same type. This is called a "nominal" type system, where two types are only equal if they have the same name, as opposed to a "structural" type system, where definitions with identical structures will be the same type regardless of what they are called.

Using records in practice

How can we use records? Let us count the ways...

Using records for function results

Just like tuples, records are useful for passing back multiple values from a function. Let's revisit the tuple examples described earlier, rewritten to use records instead:

```
// the tuple version of TryParse
let tryParseTuple intStr =
   try
      let i = System.Int32.Parse intStr
      (true,i)
   with _ -> (false,0)  // any exception

// for the record version, create a type to hold the return result
type TryParseResult = {success:bool; value:int}

// the record version of TryParse
let tryParseRecord intStr =
   try
      let i = System.Int32.Parse intStr
      {success=true;value=i}
   with _ -> {success=false;value=0}

//test it
tryParseTuple "99"
tryParseRecord "99"
tryParseTuple "abc"
tryParseRecord "abc"
```

You can see that having explicit labels in the return value makes it much easier to understand (of course, in practice we would probably use an `Option` type, discussed later).

And here's the word and letter count example using records rather than tuples:

```
//define return type
type WordAndLetterCountResult = {wordCount:int; letterCount:int}

let wordAndLetterCount (s:string) =
   let words = s.Split [|' '|]
   let letterCount = words |> Array.sumBy (fun word -> word.Length )
   {wordCount=words.Length; letterCount=letterCount}

//test
wordAndLetterCount "to be or not to be"
```

Creating records from other records

Again, as with most F# values, records are immutable and the elements within them cannot be assigned to. So how do you change a record? Again the answer is that you can't -- you must always create a new one.

Say that you need to write a function that, given a `GeoCoord` record, adds one to each element. Here it is:

```
let addOneToGeoCoord aGeoCoord =
   let {lat=x; long=y} = aGeoCoord
   {lat = x + 1.0; long = y + 1.0}   // create a new one

// try it
addOneToGeoCoord {lat=1.1; long=2.2}
```

But again you can simplify by deconstructing directly in the parameters of a function, so that the function becomes a one liner:

```
let addOneToGeoCoord {lat=x; long=y} = {lat=x+1.0; long=y+1.0}

// try it
addOneToGeoCoord {lat=1.0; long=2.0}
```

or depending on your taste, you can also use dot notation to get the properties:

```
let addOneToGeoCoord aGeoCoord =
   {lat=aGeoCoord.lat + 1.0; long= aGeoCoord.long + 1.0}
```

In many cases, you just need to tweak one or two fields and leave all the others alone. To make life easier, there is a special syntax for this common case, the " `with` " keyword. You start with the original value, followed by "with" and then the fields you want to change. Here are some examples:

```
let g1 = {lat=1.1; long=2.2}
let g2 = {g1 with lat=99.9}   // create a new one

let p1 = {first="Alice"; last="Jones"}
let p2 = {p1 with last="Smith"}
```

The technical term for "with" is a copy-and-update record expression.

Record equality

Like tuples, records have an automatically defined equality operation: two records are equal if they have the same type and the values in each slot are equal.

And records also have an automatically defined hash value based on the values in the record, so that records can be used as dictionary keys without problems.

```
{first="Alice"; last="Jones"}.GetHashCode()
```

Record representation

As noted in a previous post, records have a nice default string representation, and can be serialized easily. But unlike tuples, the `ToString()` representation is unhelpful.

```
printfn "%A" {first="Alice"; last="Jones"}   // nice
{first="Alice"; last="Jones"}.ToString()     // ugly
printfn "%O" {first="Alice"; last="Jones"}   // ugly
```

Sidebar: %A vs. %O in print format strings

We just saw that print format specifiers `%A` and `%O` produce very different results for the same record:

```
printfn "%A" {first="Alice"; last="Jones"}
printfn "%O" {first="Alice"; last="Jones"}
```

So why the difference?

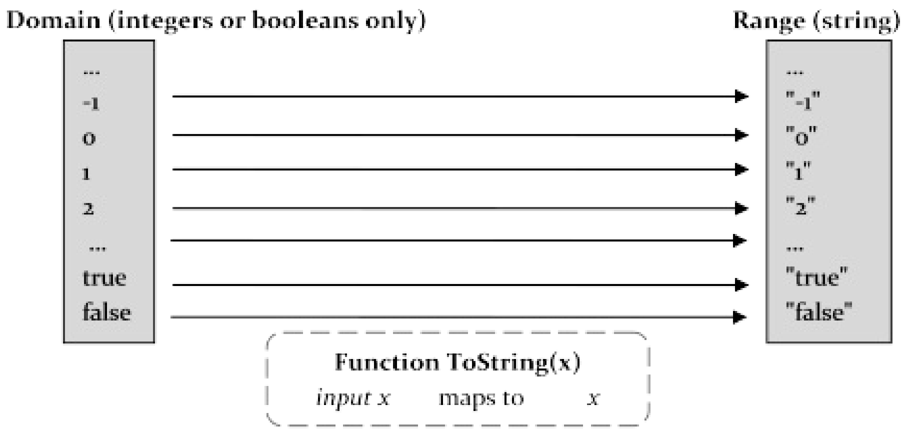
`%A` prints the value using the same pretty printer that is used for interactive output. But `%O` uses `Object.ToString()`, which means that if the `ToString` method is not overridden, `%O` will give the default (and generally unhelpful) output. So in general, you should try to use `%A` to `%O` where possible, because the core F# types do have pretty-printing by default.

But note that the F# "class" types do *not* have a standard pretty printed format, so `%A` and `%O` are equally uncooperative unless you override `ToString`.

Discriminated Unions

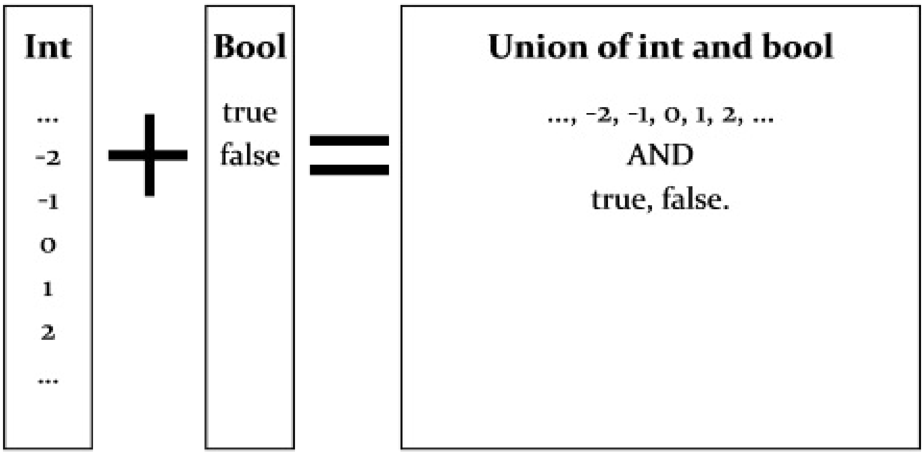
Tuples and records are examples of creating new types by "multiplying" existing types together. At the beginning of the series, I mentioned that the other way of creating new types was by "summing" existing types. What does this mean?

Well, let's say that we want to define a function that works with integers OR booleans, maybe to convert them into strings. But we want to be strict and not accept any other type (such as floats or strings). Here's a diagram of such as function:



How could we represent the domain of this function?

What we need is a type that represents all possible integers PLUS all possible booleans.



In other words, a "sum" type. In this case the new type is the "sum" of the integer type plus the boolean type.

In F#, a sum type is called a "discriminated union" type. Each component type (called a *union case*) must be tagged with a label (called a *case identifier* or *tag*) so that they can be told apart ("discriminated"). The labels can be any identifier you like, but must start with an uppercase letter.

Here's how we might define the type above:

```
type IntOrBool =
  | I of int
  | B of bool
```

The "I" and the "B" are just arbitrary labels; we could have used any other labels that were meaningful.

For small types, we can put the definition on one line:

```
type IntOrBool = I of int | B of bool
```

The component types can be any other type you like, including tuples, records, other union types, and so on.

```
type Person = {first:string; last:string}  // define a record type
type IntOrBool = I of int | B of bool

type MixedType =
  | Tup of int * int  // a tuple
  | P of Person       // use the record type defined above
  | L of int list     // a list of ints
  | U of IntOrBool    // use the union type defined above
```

You can even have types that are recursive, that is, they refer to themselves. This is typically how tree structures are defined. Recursive types will be discussed in more detail shortly.

Sum types vs. C++ unions and VB variants

At first glance, a sum type might seem similar to a union type in C++ or a variant type in Visual Basic, but there is a key difference. The union type in C++ is not type-safe and the data stored in the type can be accessed using any of the possible tags. An F# discriminated union type is safe, and the data can only be accessed one way. It really is helpful to think of it as a sum of two types (as shown in the diagram), rather than as just an overlay of data.

Key points about union types

Some key things to know about union types are:

- The vertical bar is optional before the first component, so that the following definitions are all equivalent, as you can see by examining the output of the interactive window:

```
type IntOrBool = I of int | B of bool     // without initial bar
type IntOrBool = | I of int | B of bool   // with initial bar
type IntOrBool =
   | I of int
   | B of bool      // with initial bar on separate lines
```

- The tags or labels must start with an uppercase letter. So the following will give an error:

```
type IntOrBool = int of int| bool of bool
//  error FS0053: Discriminated union cases
//                must be uppercase identifiers
```

- Other named types (such as `Person` or `IntOrBool`) must be pre-defined outside the union type. You can't define them "inline" and write something like this:

```
type MixedType =
  | P of  {first:string; last:string}  // error
```

or

```
type MixedType =
  | U of (I of int | B of bool)  // error
```

- The labels can be any identifier, including the names of the component type themselves, which can be quite confusing if you are not expecting it. For example, if the `Int32` and `Boolean` types (from the `System` namespace) were used instead, and the labels were named the same, we would have this perfectly valid definition:

```
open System
type IntOrBool = Int32 of Int32 | Boolean of Boolean
```

This "duplicate naming" style is actually quite common, because it documents exactly what the component types are.

Constructing a value of a union type

To create a value of a union type, you use a "constructor" that refers to only one of the possible union cases. The constructor then follows the form of the definition, using the case label as if it were a function. In the `IntOrBool` example, you would write:

```
type IntOrBool = I of int | B of bool

let i  = I 99    // use the "I" constructor
// val i : IntOrBool = I 99

let b  = B true  // use the "B" constructor
// val b : IntOrBool = B true
```

The resulting value is printed out with the label along with the component type:

```
val [value name] : [type]    = [label] [print of component type]
val i            : IntOrBool = I       99
val b            : IntOrBool = B       true
```

If the case constructor has more than one "parameter", you construct it in the same way that you would call a function:

```
type Person = {first:string; last:string}

type MixedType =
  | Tup of int * int
  | P of Person

let myTup  = Tup (2,99)    // use the "Tup" constructor
// val myTup : MixedType = Tup (2,99)

let myP  = P {first="Al"; last="Jones"} // use the "P" constructor
// val myP : MixedType = P {first = "Al";last = "Jones";}
```

The case constructors for union types are normal functions, so you can use them anywhere a function is expected. For example, in `List.map` :

```
type C = Circle of int | Rectangle of int * int

[1..10]
|> List.map Circle

[1..10]
|> List.zip [21..30]
|> List.map Rectangle
```

Naming conflicts

If a particular case has a unique name, then the type to construct will be unambiguous.

But what happens if you have two types which have cases with the same labels?

```
type IntOrBool1 = I of int | B of bool
type IntOrBool2 = I of int | B of bool
```

In this case, the last one defined is generally used:

```
let x = I 99                // val x : IntOrBool2 = I 99
```

But it is much better to explicitly qualify the type, as shown:

```
let x1 = IntOrBool1.I 99    // val x1 : IntOrBool1 = I 99
let x2 = IntOrBool2.B true  // val x2 : IntOrBool2 = B true
```

And if the types come from different modules, you can use the module name as well:

```
module Module1 =
  type IntOrBool = I of int | B of bool

module Module2 =
  type IntOrBool = I of int | B of bool

module Module3 =
  let x = Module1.IntOrBool.I 99 // val x : Module1.IntOrBool = I 99
```

Matching on union types

For tuples and records, we have seen that "deconstructing" a value uses the same model as constructing it. This is also true for union types, but we have a complication: which case should we deconstruct?

This is exactly what the "match" expression is designed for. As you should now realize, the match expression syntax has parallels to how a union type is defined.

```
// definition of union type
type MixedType =
  | Tup of int * int
  | P of Person

// "deconstruction" of union type
let matcher x =
  match x with
  | Tup (x,y) ->
        printfn "Tuple matched with %i %i" x y
  | P {first=f; last=l} ->
        printfn "Person matched with %s %s" f l

let myTup = Tup (2,99)                 // use the "Tup" constructor
matcher myTup

let myP = P {first="Al"; last="Jones"} // use the "P" constructor
matcher myP
```

Let's analyze what is going on here:

- Each "branch" of the overall match expression is a pattern expression that is designed to match the corresponding case of the union type.
- The pattern starts with the tag for the particular case, and then the rest of the pattern deconstructs the type for that case in the usual way.
- The pattern is followed by an arrow "->" and then the code to execute.

Empty cases

The label for a union case does not have to have to have any type after it. The following are all valid union types:

```
type Directory =
  | Root                   // no need to name the root
  | Subdirectory of string // other directories need to be named

type Result =
  | Success                // no string needed for success state
  | ErrorMessage of string // error message needed
```

If *all* the cases are empty, then we have an "enum style" union:

```
type Size = Small | Medium | Large
type Answer = Yes | No | Maybe
```

Note that this "enum style" union is *not* the same as a true C# enum type, discussed later.

To create an empty case, just use the label as a constructor without any parameters:

```
let myDir1 = Root
let myDir2 = Subdirectory "bin"

let myResult1 = Success
let myResult2 = ErrorMessage "not found"

let mySize1 = Small
let mySize2 = Medium
```

Single cases

Sometimes it is useful to create union types with only one case. This might be seem useless, because you don't seem to be adding value. But in fact, this a very useful practice that can enforce type safety*.

* And in a future series we'll see that, in conjuction with module signatures, single case unions can also help with data hiding and capability based security.

For example, let's say that we have customer ids and order ids which are both represented by integers, but that they should never be assigned to each other.

As we saw before, a type alias approach will not work, because an alias is just a synonym and doesn't create a distinct type. Here's how you might try to do it with aliases:

```
type CustomerId = int   // define a type alias
type OrderId = int      // define another type alias

let printOrderId (orderId:OrderId) =
   printfn "The orderId is %i" orderId

//try it
let custId = 1          // create a customer id
printOrderId custId   // Uh-oh!
```

But even though I explicitly annotated the `orderId` parameter to be of type `OrderId`, I can't ensure that customer ids are not accidentally passed in.

On the other hand, if we create simple union types, we can easily enforce the type distinctions.

```
type CustomerId = CustomerId of int   // define a union type
type OrderId = OrderId of int         // define another union type

let printOrderId (OrderId orderId) =  // deconstruct in the param
   printfn "The orderId is %i" orderId

//try it
let custId = CustomerId 1             // create a customer id
printOrderId custId                   // Good! A compiler error now.
```

This approach is feasible in C# and Java as well, but is rarely used because of the overhead of creating and managing the special classes for each type. In F# this approach is lightweight and therefore quite common.

A convenient thing about single case union types is you can pattern match directly against a value without having to use a full `match-with` expression.

```
// deconstruct in the param
let printCustomerId (CustomerId customerIdInt) =
   printfn "The CustomerId is %i" customerIdInt

// or deconstruct explicitly through let statement
let printCustomerId2 custId =
   let (CustomerId customerIdInt) = custId  // deconstruct here
   printfn "The CustomerId is %i" customerIdInt

// try it
let custId = CustomerId 1             // create a customer id
printCustomerId custId
printCustomerId2 custId
```

But a common "gotcha" is that in some cases, the pattern match must have parens around it, otherwise the compiler will think you are defining a function!

```
let custId = CustomerId 1
let (CustomerId customerIdInt) = custId  // Correct pattern matching
let CustomerId customerIdInt = custId    // Wrong! New function?
```

Similarly, if you ever do need to create an enum-style union type with a single case, you will have to start the case with a vertical bar in the type definition; otherwise the compiler will think you are creating an alias.

```
type TypeAlias = A     // type alias!
type SingleCase = | A   // single case union type
```

Union equality

Like other core F# types, union types have an automatically defined equality operation: two unions are equal if they have the same type and the same case and the values for that case is equal.

```
type Contact = Email of string | Phone of int

let email1 = Email "bob@example.com"
let email2 = Email "bob@example.com"

let areEqual = (email1=email2)
```

Union representation

Union types have a nice default string representation, and can be serialized easily. But unlike tuples, the ToString() representation is unhelpful.

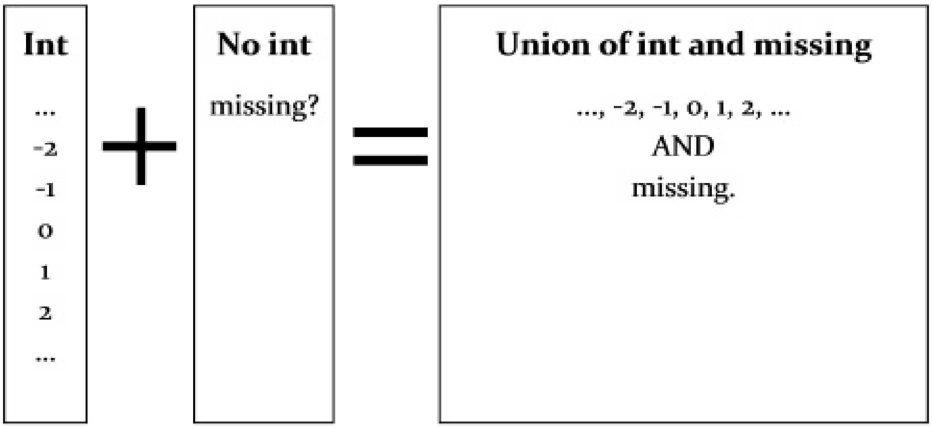
```
type Contact = Email of string | Phone of int
let email = Email "bob@example.com"
printfn "%A" email    // nice
printfn "%O" email    // ugly!
```

The Option type

Now let's look at a particular union type, the Option type. It is so common and so useful that it is actually built into the language.

You have already seen the option type discussed in passing, but let's go back to basics and understand how it fits into the type system.

A very common situation is when you want to represent missing or invalid values. Using a diagram, the domain would look like this:



Obviously this calls for some kind of union type!

In F#, it is called the `Option` type, and is defined as union type with two cases: `Some` and `None`. A similar type is common in functional languages: OCaml and Scala also call it `Option`, while Haskell calls it `Maybe`.

Here is a definition:

```
type Option<'a> =       // use a generic definition
   | Some of 'a           // valid value
   | None                 // missing
```

> IMPORTANT: if you evaluate this in the interactive window, be sure to reset the session afterwards, so that the built-in type is restored.

The option type is used in the same way as any union type in construction, by specifying one of the two cases, the `Some` case or the `None` case:

```
let validInt = Some 1
let invalidInt = None
```

and when pattern matching, as with any union type, you must always match all the cases:

```
match validInt with
| Some x -> printfn "the valid value is %A" x
| None -> printfn "the value is None"
```

When defining a type that references the Option type, you must specify the generic type to use. You can do this in an explicit way, with angle brackets, or use the built-in " `option` " keyword which comes after the type. The following examples are identical:

```
type SearchResult1 = Option<string>  // Explicit C#-style generics
type SearchResult2 = string option   // built-in postfix keyword
```

Using the Option type

The option type is widely used in the F# libraries for values that might be missing or otherwise invalid.

For example, the `List.tryFind` function returns an option, with the `None` case used indicate that nothing matches the search predicate.

```
[1;2;3;4]  |> List.tryFind (fun x-> x = 3)  // Some 3
[1;2;3;4]  |> List.tryFind (fun x-> x = 10) // None
```

Let's revisit the same example we used for tuples and records, and see how options might be used instead:

```
// the tuple version of TryParse
let tryParseTuple intStr =
   try
      let i = System.Int32.Parse intStr
      (true,i)
   with _ -> (false,0)  // any exception

// for the record version, create a type to hold the return result
type TryParseResult = {success:bool; value:int}

// the record version of TryParse
let tryParseRecord intStr =
   try
      let i = System.Int32.Parse intStr
      {success=true;value=i}
   with _ -> {success=false;value=0}

// the option version of TryParse
let tryParseOption intStr =
   try
      let i = System.Int32.Parse intStr
      Some i
   with _ -> None

//test it
tryParseTuple "99"
tryParseRecord "99"
tryParseOption "99"
tryParseTuple "abc"
tryParseRecord "abc"
tryParseOption "abc"
```

Of these three approaches, the "option" version is generally preferred; no new types need to be defined and for simple cases, the meaning of `None` is obvious from the context.

NOTE: The `tryParseOption` code is just an example. A similar function `tryParse` is built into the .NET core libraries and should be used instead.

Option equality

Like other union types, option types have an automatically defined equality operation

```
let o1 = Some 42
let o2 = Some 42

let areEqual = (o1=o2)
```

Option representation

Option types have a nice default string representation, and unlike other union types, the `ToString()` representation is also nice.

```
let o = Some 42
printfn "%A" o   // nice
printfn "%O" o   // nice
```

Options are not just for primitive types

The F# option is a true first class type (it's just a normal union type, after all). You can use it with *any* type. For example, you can have an option of a complex type like Person, or a tuple type like `int*int`, or a function type like `int->bool`, or even an option of an option type.

```
type OptionalString = string option
type OptionalPerson = Person option       // optional complex type
type OptionalTuple = (int*int) option
type OptionalFunc = (int -> bool) option  // optional function
type NestedOptionalString = OptionalString option //nested options!
type StrangeOption = string option option option
```

How the Option type should not be used

The option type has functions such as `IsSome`, `IsNone` and `Value`, which allow you to access the "wrapped" value without doing pattern matching. Don't use them! Not only it is not idiomatic, but it is dangerous and can cause exceptions.

Here is how not to do it:

```
let x = Some 99

// testing using IsSome
if x.IsSome then printfn "x is %i" x.Value   // ugly!!

// no matching at all
printfn "x is %i" x.Value   // ugly and dangerous!!
```

Here is how to do it properly:

```
let x = Some 99
match x with
| Some i -> printfn "x is %i" i
| None -> () // what to do here?
```

The pattern matching approach also forces you to think about and document what happens in the `None` case, which you might easily overlook when using `IsSome`.

The Option module

If you are doing a lot of pattern matching on options, look into the `Option` module, as it has some useful helper functions like `map`, `bind`, `iter` and so on.

For example, say that I want to multiply the value of an option by 2 if it is valid. Here's the pattern matching way:

```
let x = Some 99
let result = match x with
| Some i -> Some(i * 2)
| None -> None
```

And here's a more compact version written using `Option.map`:

```
let x = Some 99
x |> Option.map (fun v -> v * 2)
```

Or perhaps I want to multiply the value of an option by 2 if it is valid but return 0 if it is `None`. Here's the pattern matching way:

```
let x = Some 99
let result = match x with
| Some i -> i * 2
| None -> 0
```

And here's the same thing as a one-liner using `Option.fold`:

```
let x = Some 99
x |> Option.fold (fun _ v -> v * 2) 0
```

In simple cases like the one above, the `defaultArg` function can be used as well.

```
let x = Some 99
defaultArg x 0
```

Option vs. Null vs. Nullable

The option type often causes confusion to people who are used to dealing with nulls and nullables in C# and other languages. This section will try to clarify the differences.

Type safety of Option vs. null

In a language like C# or Java, "null" means a reference or pointer to an object that doesn't exist. The "null" has *exactly the same type* as the object, so you can't tell from the type system that you have a null.

For example, in the C# code below we create two string variables, one with a valid string and one with a null string.

```
string s1 = "abc";
var len1 = s1.Length;

string s2 = null;
var len2 = s2.Length;
```

This compiles perfectly, of course. The compiler cannot tell the difference between the two variables. The `null` is exactly the same type as the valid string, so all the `System.String` methods and properties can be used on it, including the `Length` property.

Now, we know that this code will fail by just looking at it, but the compiler can't help us. Instead, as we all know, you have to tediously test for nulls constantly.

Now let's look at the nearest F# equivalent of the C# example above. In F#, to indicate missing data, you would use an option type and set it to `None`. (In this artificial example we have to use an ugly explicitly typed `None` -- normally this would not be necessary.)

```
let s1 = "abc"
var len1 = s1.Length

// create a string option with value None
let s2 = Option<string>.None
let len2 = s2.Length
```

In the F# version, we get a *compile-time* error immediately. The `None` is *not* a string, it's a different type altogether, so you can't call `Length` on it directly. And to be clear, `Some [string]` is *also* not the same type as `string`, so you can't call `Length` on it either!

So if `Option<string>` is not a string, but you want to do something with the string it (might) contain, you are forced to have to pattern match on it (assuming you don't do bad things as described earlier).

```
let s2 = Option<string>.None

//which one is it?
let len2 = match s2 with
| Some s -> s.Length
| None -> 0
```

You always have to pattern match, because given a value of type `Option<string>`, you can't tell whether it is Some or None.

In just the same way `Option<int>` is not the same type as `int`, `Option<bool>` is not the same type as `bool`, and so on.

To summarize the critical points:

- The type "`string option`" is not at all the same type as "`string`". You cannot cast from `string option` to `string` -- they do not have the same properties. A function that works with `string` will not work with `string option`, and vice versa. So the type system will prevent any errors.
- On the other hand, a "null string" in C# is exactly the same type as "string". You cannot tell them apart at compile time, only at run time. A "null string" appears to have all the same properties and functions as a valid string, except that your code will blow up when you try to use it!

Nulls vs. missing data

A "null" as used in C# is completely different from the concept of "missing" data, which is a valid part of modeling any system in any language.

In a true functional language there can be a concept of missing data, but there can be no such thing as "null", because the concepts of "pointers" or "uninitialized variables" do not exist in the functional way of thinking.

For example, consider a value bound to the result of an expression like this:

```
let x = "hello world"
```

How can that value ever be uninitialized, or become null, or even become any other value at all?

Unfortunately, additional confusion has been caused because in some cases API designers have used null to indicate the concept of "missing" data as well! For example, the .NET library method `StreamReader.ReadLine` returns null to indicate that there is no more data in a file.

F# and null

F# is not a pure functional language, and has to interact with the .NET languages that *do* have the concept of null. Therefore, F# does include a `null` keyword in its design, but makes it hard to use and treats it as an abnormal value.

As a general rule, nulls are never created in "pure" F#, but only by interacting with the .NET libraries or other external systems.

Here are some examples:

```
// pure F# type is not allowed to be null (in general)
type Person = {first:string; last:string}
let p : Person = null                      // error!

// type defined in CLR, so is allowed to be null
let s : string = null                      // no error!
let line = streamReader.ReadLine()         // no error if null
```

In these cases, it is good practice to immediately check for nulls and convert them into an option type!

```
// streamReader example
let line = match streamReader.ReadLine()  with
           | null -> None
           | line -> Some line

// environment example
let GetEnvVar var =
    match System.Environment.GetEnvironmentVariable(var) with
    | null -> None
    | value -> Some value

// try it
GetEnvVar "PATH"
GetEnvVar "TEST"
```

And on occasion, you may need to pass a null to an external library. You can do this using the `null` keyword as well.

Option vs. Nullable

In addition to null, C# has the concept of a Nullable type, such as `Nullable<int>`, which seems similar to the option type. So what's the difference?

The basic idea is the same, but Nullable is much weaker. It only works on value types such as `Int` and `DateTime`, not on reference types such as strings or classes or functions. You can't nest Nullables, and they don't have much special behavior.

On the other hand, the F# option is a true first class type and can be used consistently across all types in the same way. (See the examples above in the "Options are not just for primitive types" section.)

Enum types

The enum type in F# is the same as the enum type in C#. Its definition is superficially just like that of a union type, but there are many non-obvious differences to be aware of.

Defining enums

To define an enum you use exactly the same syntax as a union type with empty cases, except that you must specify a constant value for each case, and the constants must all be of the same type.

```
type SizeUnion = Small | Medium | Large         // union
type ColorEnum = Red=0 | Yellow=1 | Blue=2      // enum
```

Strings are not allowed, only ints or compatible types such bytes and chars:

```
type MyEnum = Yes = "Y" | No ="N"  // Error. Strings not allowed.
type MyEnum = Yes = 'Y' | No ='N'  // Ok because char was used.
```

Union types require that their cases start with an uppercase letter. This is not required for enums.

```
type SizeUnion = Small | Medium | large      // Error - "large" is invalid.
type ColorEnum = Red=0 | Yellow=1 | blue=2      // Ok
```

Just as with C#, you can use the FlagsAttribute for bit flags:

```
[<System.FlagsAttribute>]
type PermissionFlags = Read = 1 | Write = 2 | Execute = 4
let permission = PermissionFlags.Read ||| PermissionFlags.Write
```

Constructing enums

Unlike union types, to construct an enum you *must always* use a qualified name:

```
let red = Red             // Error. Enums must be qualified
let red = ColorEnum.Red  // Ok
let small = Small         // Ok.  Unions do not need to be qualified
```

You can also cast to and from the underlying int type:

```
let redInt = int ColorEnum.Red
let redAgain:ColorEnum = enum redInt // cast to a specified enum type
let yellowAgain = enum<ColorEnum>(1) // or create directly
```

You can even create values that are not on the enumerated list at all.

```
let unknownColor = enum<ColorEnum>(99)   // valid
```

And, unlike unions, you can use the BCL Enum functions to enumerate and parse values, just as with C#. For example:

```
let values = System.Enum.GetValues(typeof<ColorEnum>)
let redFromString =
    System.Enum.Parse(typeof<ColorEnum>,"Red")
    :?> ColorEnum  // downcast needed
```

Matching enums

To match an enum you must again *always* use a qualified name:

```
let unqualifiedMatch x =
    match x with
    | Red -> printfn "red"             // warning FS0049
    | _ -> printfn "something else"

let qualifiedMatch x =
    match x with
    | ColorEnum.Red -> printfn "red"   //OK. qualified name used.
    | _ -> printfn "something else"
```

Both unions and enums will warn if you have not covered all known cases when pattern matching:

```
let matchUnionIncomplete x =
    match x with
    | Small -> printfn "small"
    | Medium -> printfn "medium"
    // Warning: Incomplete pattern matches

let matchEnumIncomplete x =
    match x with
    | ColorEnum.Red -> printfn "red"
    | ColorEnum.Yellow -> printfn "yellow"
    // Warning: Incomplete pattern matches
```

One important difference between unions and enums is that can you make the compiler happy about exhaustive pattern matching by listing all the union types.

Not so for enums. It is possible to create an enum not on the predeclared list, and try to match with it, and get a runtime exception, so the compiler will warn you even if you have explicitly listed all the known enums:

```
// the compiler is still not happy
let matchEnumIncomplete2 x =
    match x with
    | ColorEnum.Red -> printfn "red"
    | ColorEnum.Yellow -> printfn "yellow"
    | ColorEnum.Blue -> printfn "blue"
    // the value '3' may indicate a case not covered by the pattern(s).
```

The only way to fix this is to add a wildcard to the bottom of the cases, to handle enums outside the predeclared range.

```
// the compiler is finally happy
let matchEnumComplete x =
    match x with
    | ColorEnum.Red -> printfn "red"
    | ColorEnum.Yellow -> printfn "yellow"
    | ColorEnum.Blue -> printfn "blue"
    | _ -> printfn "something else"

// test with unknown case
let unknownColor = enum<ColorEnum>(99)   // valid
matchEnumComplete unknownColor
```

Summary

In general, you should prefer discriminated union types over enums, unless you really need to have an `int` value associated with them, or you are writing types that need to be exposed to other .NET languages.

Built-in .NET types

In this post we'll take a quick look at how F# handles the standard types that are built into .NET.

Literals

F# uses the same syntax for literals that C# does, with a few exceptions.

I'll divide the built-in types into the following groups:

- miscellaneous types (`bool` , `char` , etc.)
- string types
- integer types (`int` , `uint` and `byte` , etc)
- float types (`float` , `decimal` , etc)
- pointer types (`IntPtr` , etc)

The following tables list the primitive types, with their F# keywords, their suffixes if any, an example, and the corresponding .NET CLR type.

Miscellaneous types

	Object	Unit	Bool	Char (Unicode)	Char (Ascii)
Keyword	obj	unit	bool	char	byte
Suffix					B
Example	let o = obj()	let u = ()	true false	'a'	'a'B
.NET Type	Object	(no equivalent)	Boolean	Char	Byte

Object and unit are not really .NET primitive types, but I have included them for the sake of completeness.

String types

	String (Unicode)	Verbatim string (Unicode)	Triple quoted string (Unicode)	String (Ascii)
Keyword	string	string	string	byte[]
Suffix				
Example	"first\nsecond line"	@"C:\name"	"""can "contain"" special chars"""	"aaa"B
.NET Type	String	String	String	Byte[]

The usual special characters can be used inside normal strings, such as `\n` , `\t` , `\\` , etc. Quotes must be escaped with a backslash: `\'` and `\"` .

In verbatim strings, backslashes are ignored (good for Windows filenames and regex patterns). But quotes need to be doubled.

Triple-quoted strings are new in VS2012. They are useful because special characters do not need to be escaped at all, and so they can handle embedded quotes nicely (great for XML).

Integer types

	8 bit (Signed)	8 bit (Unsigned)	16 bit (Signed)	16 bit (Unsigned)	32 bit (Signed)	32 bi (Unsigr
Keyword	sbyte	byte	int16	uint16	int	uint32
Suffix	y	uy	s	us		u
Example	99y	99uy	99s	99us	99	99u
.NET Type	SByte	Byte	Int16	UInt16	Int32	UInt32

`BigInteger` is available in all versions of F#. From .NET 4 it is included as part of the .NET base library.

Integer types can also be written in hex and octal.

- The hex prefix is `0x` . So `0xFF` is hex for 255.
- The octal prefix is `0o` . So `0o377` is octal for 255.

Floating point types

	32 bit floating point	64 bit (default) floating point	High precision floating point
Keyword	float32, single	float, double	decimal
Suffix	f		m
Example	123.456f	123.456	123.456m
.NET Type	Single	Double	Decimal

Note that F# natively uses `float` instead of `double`, but both can be used.

Pointer types

	Pointer/handle (signed)	Pointer/handle (unsigned)
Keyword	nativeint	unativeint
Suffix	n	un
Example	0xFFFFFFFFn	0xFFFFFFFFun
.NET Type	IntPtr	UIntPtr

Casting between built-in primitive types

Note: this section only covers casting of primitive types. For casting between classes see the series on object-oriented programming.

There is no direct "cast" syntax in F#, but there are helper functions to cast between types. These helper functions have the same name as the type (you can see them in the `Microsoft.FSharp.Core` namespace).

So for example, in C# you might write:

```
var x = (int)1.23
var y = (double)1
```

In F# the equivalent would be:

```
let x = int 1.23
let y = float 1
```

In F# there are only casting functions for numeric types. In particular, there is no cast for bool, and you must use `Convert` or similar.

```
let x = bool 1  //error
let y = System.Convert.ToBoolean(1)  // ok
```

Boxing and unboxing

Just as in C# and other .NET languages, the primitive int and float types are value objects, not classes. Although this is normally transparent, there are certain occasions where it can be an issue.

First, lets look at the transparent case. In the example below, we define a function that takes a parameter of type `Object`, and simply returns it. If we pass in an `int`, it is silently boxed into an object, as can be seen from the test code, which returns an `object` not an `int`.

```
// create a function with parameter of type Object
let objFunction (o:obj) = o

// test: call with an integer
let result = objFunction 1

// result is
// val result : obj = 1
```

The fact that `result` is an object, not an int, can cause type errors if you are not careful. For example, the result cannot be directly compared with the original value:

```
let resultIsOne = (result = 1)
// error FS0001: This expression was expected to have type obj
// but here has type int
```

To work with this situation, and other similar ones, you can convert a primitive type to an object directly, by using the `box` keyword:

```
let o = box 1

// retest the comparison example above, but with boxing
let result = objFunction 1
let resultIsOne = (result = box 1)  // OK
```

To convert an object back to an primitive type, use the `unbox` keyword, but unlike `box`, you must either supply a specific type to unbox to, or be sure that the compiler has enough information to make an accurate type inference.

```
// box an int
let o = box 1

// type known for target value
let i:int = unbox o  // OK

// explicit type given in unbox
let j = unbox<int> o  // OK

// type inference, so no type annotation needed
let k = 1 + unbox o  // OK
```

So the comparison example above could also be done with `unbox`. No explicit type annotation is needed because it is being compared with an int.

```
let result = objFunction 1
let resultIsOne = (unbox result = 1)  // OK
```

A common problem occurs if you do not specify enough type information -- you will encounter the infamous "Value restriction" error, as shown below:

```
let o = box 1

// no type specified
let i = unbox o  // FS0030: Value restriction error
```

The solution is to reorder the code to help the type inference, or when all else fails, add an explicit type annotation. See the post on type inference for more tips.

Boxing in combination with type detection

Let's say that you want to have a function that matches based on the type of the parameter, using the `:?` operator:

```
let detectType v =
    match v with
        | :? int -> printfn "this is an int"
        | _ -> printfn "something else"
```

Unfortunately, this code will fail to compile, with the following error:

```
// error FS0008: This runtime coercion or type test from type 'a to int
// involves an indeterminate type based on information prior to this program point.
// Runtime type tests are not allowed on some types. Further type annotations are need
ed.
```

The message tells you the problem: "runtime type tests are not allowed on some types".

The answer is to "box" the value which forces it into a reference type, and then you can type check it:

```
let detectTypeBoxed v =
    match box v with      // used "box v"
        | :? int -> printfn "this is an int"
        | _ -> printfn "something else"

//test
detectTypeBoxed 1
detectTypeBoxed 3.14
```

Units of measure

As we mentioned earlier in the "why use F#?" series, F# has a very cool feature which allows you to add extra unit-of-measure information to as metadata to numeric types.

The F# compiler will then make sure that only numerics with the same unit-of-measure can be combined. This can be very useful to stop accidental mismatches and to make your code safer.

Defining units of measure

A unit of measure definition consists of the attribute `[<Measure>]`, followed by the `type` keyword and then a name. For example:

```
[<Measure>]
type cm

[<Measure>]
type inch
```

Often you will see the whole definition written on one line instead:

```
[<Measure>] type cm
[<Measure>] type inch
```

Once you have a definition, you can associate a measure type with a numeric type by using angle brackets with measure name inside:

```
let x = 1<cm>    // int
let y = 1.0<cm>  // float
let z = 1.0m<cm> // decimal
```

You can even combine measures within the angle brackets to create compound measures:

```
[<Measure>] type m
[<Measure>] type sec
[<Measure>] type kg

let distance = 1.0<m>
let time = 2.0<sec>
let speed = 2.0<m/sec>
let acceleration = 2.0<m/sec^2>
let force = 5.0<kg m/sec^2>
```

Derived units of measure

If you use certain combinations of units a lot, you can define a *derived* measure and use that instead.

```
[<Measure>] type N = kg m/sec^2

let force1 = 5.0<kg m/sec^2>
let force2 = 5.0<N>

force1 = force2 // true
```

SI units and constants

If you are using the units-of-measure for physics or other scientific applications, you will definitely want to use the SI units and related constants. You don't need to define all these yourself! These are predefined for you and available as follows:

- In F# 3.0 and higher (which shipped with Visual Studio 2012), these are built into the core F# libraries in the `Microsoft.FSharp.Data.UnitSystems.SI` namespace (see the MSDN page).
- In F# 2.0 (which shipped with Visual Studio 2010), you will have to install the F# powerpack to get them. (The F# powerpack is on Codeplex at http://fsharppowerpack.codeplex.com).

Type checking and type inference

The units-of-measure are just like proper types; you get static checking *and* type inference.

```
[<Measure>] type foot
[<Measure>] type inch

let distance = 3.0<foot>

// type inference for result
let distance2 = distance * 2.0

// type inference for input and output
let addThreeFeet ft =
    ft + 3.0<foot>
```

And of course, when using them, the type checking is strict:

```
addThreeFeet 1.0        //error
addThreeFeet 1.0<inch>  //error
addThreeFeet 1.0<foot>  //OK
```

Type annotations

If you want to be explicit in specifying a unit-of-measure type annotation, you can do so in the usual way. The numeric type must have angle brackets with the unit-of-measure.

```
let untypedTimesThree (ft:float) =
    ft * 3.0

let footTimesThree (ft:float<foot>) =
    ft * 3.0
```

Combining units of measure with multiplication and division

The compiler understands how units of measure transform when individual values are multiplied or divided.
For example, in the following, the `speed` value has been automatically given the measure `<m/sec>`.

```
[<Measure>] type m
[<Measure>] type sec
[<Measure>] type kg

let distance = 1.0<m>
let time = 2.0<sec>
let speed = distance/time
let acceleration = speed/time
let mass = 5.0<kg>
let force = mass * speed/time
```

Look at the types of the `acceleration` and `force` values above to see other examples of how this works.

Dimensionless values

A numeric value without any specific unit of measure is called *dimensionless*. If you want to be explicit that a value is dimensionless, you can use the measure called `1`.

```
// dimensionless
let x = 42

// also dimensionless
let x = 42<1>
```

Mixing units of measure with dimensionless values

Note that you cannot *add* a dimensionless value to a value with a unit of measure, but you can *multiply or divide* by dimensionless values.

```
// test addition
3.0<foot> + 2.0<foot>  // OK
3.0<foot> + 2.0        // error

// test multiplication
3.0<foot> * 2.0        // OK
```

But see the section on "generics" below for an alternative approach.

Conversion between units

What if you need to convert between units?

It's straightforward. You first need to define a conversion value that uses *both* units, and then multiply the source value by the conversion factor.

Here's an example with feet and inches:

```
[<Measure>] type foot
[<Measure>] type inch

//conversion factor
let inchesPerFoot = 12.0<inch/foot>

// test
let distanceInFeet = 3.0<foot>
let distanceInInches = distanceInFeet * inchesPerFoot
```

And here's an example with temperature:

```
[<Measure>] type degC
[<Measure>] type degF

let convertDegCToF c =
    c * 1.8<degF/degC> + 32.0<degF>

// test
let f = convertDegCToF 0.0<degC>
```

The compiler correctly inferred the signature of the conversion function.

```
val convertDegCToF : float<degC> -> float<degF>
```

Note that the constant `32.0<degF>` was explicitly annotated with the `degF` so that the result would be in `degF` as well. If you leave off this annotation, the result is a plain float, and the function signature changes to something much stranger! Try it and see:

```
let badConvertDegCToF c =
    c * 1.8<degF/degC> + 32.0
```

Conversion between dimensionless values and unit-of-measure values

To convert from a dimensionless numeric value to a value with a measure type, just multiply it by one, but with the one annotated with the appropriate unit.

```
[<Measure>] type foot

let ten = 10.0   // normal

//converting from non-measure to measure
let tenFeet = ten * 1.0<foot>  // with measure
```

And to convert the other way, either divide by one, or multiply with the inverse unit.

```
//converting from measure to non-measure
let tenAgain = tenFeet / 1.0<foot>  // without measure
let tenAnotherWay = tenFeet * 1.0<1/foot>  // without measure
```

The above methods are type safe, and will cause errors if you try to convert the wrong type.

If you don't care about type checking, you can do the conversion with the standard casting functions instead:

```
let tenFeet = 10.0<foot>  // with measure
let tenDimensionless = float tenFeet // without measure
```

Generic units of measure

Often, we want to write functions that will work with any value, no matter what unit of measure is associated with it.

For example, here is our old friend `square`. But when we try to use it with a unit of measure, we get an error.

```
let square x = x * x

// test
square 10<foot>   // error
```

What can we do? We don't want to specify a particular unit of measure, but on the other hand we must specify *something*, because the simple definition above doesn't work.

The answer is to use *generic* units of measure, indicated with an underscore where the measure name normally is.

```
let square (x:int<_>) = x * x

// test
square 10<foot>   // OK
square 10<sec>   // OK
```

Now the `square` function works as desired, and you can see that the function signature has used the letter `'u` to indicate a generic unit of measure. And also note that the compiler has inferred that the return value is of type "unit squared".

```
val square : int<'u> -> int<'u ^ 2>
```

Indeed, you can specify the generic type using letters as well if you like:

```
// with underscores
let square (x:int<_>) = x * x

// with letters
let square (x:int<'u>) = x * x

// with underscores
let speed (distance:float<_>) (time:float<_>) =
    distance / time

// with letters
let speed (distance:float<'u>) (time:float<'v>) =
    distance / time
```

You may need to use letters sometimes to explicitly indicate that the units are the same:

```
let ratio (distance1:float<'u>) (distance2:float<'u>) =
    distance1 / distance2
```

Using generic measures with lists

You cannot always use a measure directly. For example, you cannot define a list of feet directly:

```
//error
[1.0<foot>..10.0<foot>]
```

Instead, you have to use the "multiply by one" trick mentioned above:

```
//converting using map -- OK
[1.0..10.0] |> List.map (fun i-> i * 1.0<foot>)

//using a generator -- OK
[ for i in [1.0..10.0] -> i * 1.0<foot> ]
```

Using generic measures for constants

Multiplication by constants is OK (as we saw above), but if you try to do addition, you will get an error.

```
let x = 10<foot> + 1  // error
```

The fix is to add a generic type to the constant, like this:

```
let x = 10<foot> + 1<_>  // ok
```

A similar situation occurs when passing in constants to a higher order function such as `fold`.

```
let feet = [ for i in [1.0..10.0] -> i * 1.0<foot> ]

// OK
feet |> List.sum

// Error
feet |> List.fold (+) 0.0

// Fixed with generic 0
feet |> List.fold (+) 0.0<_>
```

Issues with generic measures with functions

There are some cases where type inference fails us. For example, let's try to create a simple `add1` function that uses units.

```
// try to define a generic function
let add1 n = n + 1.0<_>
// warning FS0064: This construct causes code to be less generic than
// indicated by the type annotations. The unit-of-measure variable 'u
// has been constrained to be measure '1'.

// test
add1 10.0<foot>
// error FS0001: This expression was expected to have type float
// but here has type float<foot>
```

The warning message has the clue. The input parameter `n` has no measure, so the measure for `1<_>` will always be ignored. The `add1` function does not have a unit of measure so when you try to call it with a value that does have a measure, you get an error.

So maybe the solution is to explicitly annotate the measure type, like this:

```
// define a function with explicit type annotation
let add1 (n:float<'u>) : float<'u> =  n + 1.0<_>
```

But no, you get the same warning FS0064 again.

Maybe we can replace the underscore with something more explicit such as `1.0<'u>` ?

```
let add1 (n:float<'u>) : float<'u> = n + 1.0<'u>
// error FS0634: Non-zero constants cannot have generic units.
```

But this time we get a compiler error!

The answer is to use one of the helpful utility functions in the LanguagePrimitives module: `FloatWithMeasure` , `Int32WithMeasure` , etc.

```
// define the function
let add1 n  =
    n + (LanguagePrimitives.FloatWithMeasure 1.0)

// test
add1 10.0<foot>   // Yes!
```

And for generic ints, you can use the same approach:

```
open LanguagePrimitives

let add2Int n  =
    n + (Int32WithMeasure 2)

add2Int 10<foot>   // OK
```

Using generic measures with type definitions

That takes care of functions. What about when we need to use a unit of measure in a type definition?

Say we want to define a generic coordinate record that works with an unit of measure. Let's start with a naive approach:

```
type Coord =
    { X: float<'u>; Y: float<'u>; }
// error FS0039: The type parameter 'u' is not defined
```

That didn't work, so what about adding the measure as a type parameter:

```
type Coord<'u> =
    { X: float<'u>; Y: float<'u>; }
// error FS0702: Expected unit-of-measure parameter, not type parameter.
// Explicit unit-of-measure parameters must be marked with the [<Measure>] attribute.
```

That didn't work either, but the error message tells us what to do. Here is the final, correct version, using the `Measure` attribute:

```
type Coord<[<Measure>] 'u> =
    { X: float<'u>; Y: float<'u>; }

// Test
let coord = {X=10.0<foot>; Y=2.0<foot>}
```

In some cases, you might need to define more than one measure. In the following example, the currency exchange rate is defined as the ratio of two currencies, and so needs two generic measures to be defined.

```
type CurrencyRate<[<Measure>]'u, [<Measure>]'v> =
    { Rate: float<'u/'v>; Date: System.DateTime}

// test
[<Measure>] type EUR
[<Measure>] type USD
[<Measure>] type GBP

let mar1 = System.DateTime(2012,3,1)
let eurToUsdOnMar1 = {Rate= 1.2<USD/EUR>; Date=mar1 }
let eurToGbpOnMar1 = {Rate= 0.8<GBP/EUR>; Date=mar1 }

let tenEur = 10.0<EUR>
let tenEurInUsd = eurToUsdOnMar1.Rate * tenEur
```

And of course, you can mix regular generic types with unit of measure types.

For example, a product price might consist of a generic product type, plus a price with a currency:

```
type ProductPrice<'product, [<Measure>] 'currency> =
    { Product: 'product; Price: float<'currency>; }
```

Units of measure at runtime

An issue that you may run into is that units of measure are not part of the .NET type system.

F# does stores extra metadata about them in the assembly, but this metadata is only understood by F#.

This means that there is no (easy) way at runtime to determine what unit of measure a value has, nor any way to dynamically assign a unit of measure at runtime.

It also means that there is no way to expose units of measure as part of a public API to another .NET language (except other F# assemblies).

Understanding type inference

Before we finish with types, let's revisit type inference: the magic that allows the F# compiler to deduce what types are used and where. We have seen this happen through all the examples so far, but how does it work and what can you do if it goes wrong?

How does type inference work?

It does seem to be magic, but the rules are mostly straightforward. The fundamental logic is based on an algorithm often called "Hindley-Milner" or "HM" (more accurately it should be called "Damas-Milner's Algorithm W"). If you want to know the details, go ahead and Google it.

I do recommend that you take some time to understand this algorithm so that you can "think like the compiler" and troubleshoot effectively when you need to.

Here are some of the rules for determine the types of simple and function values:

- Look at the literals
- Look at the functions and other values something interacts with
- Look at any explicit type constraints
- If there are no constraints anywhere, automatically generalize to generic types

Let's look at each of these in turn.

Look at the literals

The literals give the compiler a clue to the context. As we have seen, the type checking is very strict; ints and floats are not automatically cast to the other. The benefit of this is that the compiler can deduce types by looking at the literals. If the literal is an `int` and you are adding "x" to it, then "x" must be an int as well. But if the literal is a `float` and you are adding "x" to it, then "x" must be a float as well.

Here are some examples. Run them and see their signatures in the interactive window:

```
let inferInt x = x + 1
let inferFloat x = x + 1.0
let inferDecimal x = x + 1m     // m suffix means decimal
let inferSByte x = x + 1y       // y suffix means signed byte
let inferChar x = x + 'a'       // a char
let inferString x = x + "my string"
```

Look at the functions and other values it interacts with

If there are no literals anywhere, the compiler tries to work out the types by analyzing the functions and other values that they interact with. In the cases below, the " `indirect` " function calls a function that we do know the types for, which gives us the information to deduce the types for the " `indirect` " function itself.

```
let inferInt x = x + 1
let inferIndirectInt x = inferInt x       //deduce that x is an int

let inferFloat x = x + 1.0
let inferIndirectFloat x = inferFloat x   //deduce that x is a float
```

And of course assignment counts as an interaction too. If x is a certain type, and y is bound (assigned) to x, then y must be the same type as x.

```
let x = 1
let y = x     //deduce that y is also an int
```

Other interactions might be control structures, or external libraries

```
// if..else implies a bool
let inferBool x = if x then false else true
// for..do implies a sequence
let inferStringList x = for y in x do printfn "%s" y
// :: implies a list
let inferIntList x = 99::x
// .NET library method is strongly typed
let inferStringAndBool x = System.String.IsNullOrEmpty(x)
```

Look at any explicit type constraints or annotations

If there are any explicit type constraints or annotations specified, then the compiler will use them. In the case below, we are explicitly telling the compiler that " `inferInt2` " takes an `int` parameter. It can then deduce that the return value for " `inferInt2` " is also an `int` , which in turn implies that " `inferIndirectInt2` " is of type int->int.

```
let inferInt2 (x:int) = x
let inferIndirectInt2 x = inferInt2 x

let inferFloat2 (x:float) = x
let inferIndirectFloat2 x = inferFloat2 x
```

Note that the formatting codes in `printf` statements count as explicit type constraints too!

```
let inferIntPrint x = printf "x is %i" x
let inferFloatPrint x = printf "x is %f" x
let inferGenericPrint x = printf "x is %A" x
```

Automatic generalization

If after all this, there are no constraints found, the compiler just makes the types generic.

```
let inferGeneric x = x
let inferIndirectGeneric x = inferGeneric x
let inferIndirectGenericAgain x = (inferIndirectGeneric x).ToString()
```

It works in all directions!

The type inference works top-down, bottom-up, front-to-back, back-to-front, middle-out, anywhere there is type information, it will be used.

Consider the following example. The inner function has a literal, so we know that it returns an `int`. And the outer function has been explicitly told that it returns a `string`. But what is the type of the passed in "`action`" function in the middle?

```
let outerFn action : string =
   let innerFn x = x + 1 // define a sub fn that returns an int
   action (innerFn 2)    // result of applying action to innerFn
```

The type inference would work something like this:

- `1` is an `int`
- Therefore `x+1` must be an `int`, therefore `x` must be an `int`
- Therefore `innerFn` must be `int->int`
- Next, `(innerFn 2)` returns an `int`, therefore "`action`" takes an `int` as input.
- The output of `action` is the return value for `outerFn`, and therefore the output type of `action` is the same as the output type of `outerFn`.
- The output type of `outerFn` has been explicitly constrained to `string`, therefore the output type of `action` is also `string`.
- Putting this together, we now know that the `action` function has signature `int->string`
- And finally, therefore, the compiler deduces the type of `outerFn` as:

```
val outerFn: (int -> string) -> string
```

Elementary, my dear Watson!

The compiler can do deductions worthy of Sherlock Holmes. Here's a tricky example that will test how well you have understood everything so far.

Let's say we have a `doItTwice` function that takes any input function (call it " `f` ") and generates a new function that simply does the original function twice in a row. Here's the code for it:

```
let doItTwice f  = (f >> f)
```

As you can see, it composes `f` with itself. So in other words, it means: "do f", then "do f" on the result of that.

Now, what could the compiler possibly deduce about the signature of `doItTwice` ?

Well, let's look at the signature of " `f` " first. The output of the first call to " `f` " is also the input to the second call to " `f` ". So therefore the output and input of " `f` " must be the same type. So the signature of `f` must be `'a -> 'a` . The type is generic (written as 'a) because we have no other information about it.

So going back to `doItTwice` itself, we now know it takes a function parameter of `'a -> 'a` . But what does it return? Well, here's how we deduce it, step by step:

- First, note that `doItTwice` generates a function, so must return a function type.
- The input to the generated function is the same type as the input to first call to " `f` "
- The output of the generated function is the same type as the output of the second call to " `f` "
- So the generated function must also have type `'a -> 'a`
- Putting it all together, `doItTwice` has a domain of `'a -> 'a` and a range of `'a -> 'a` , so therefore its signature must be `('a -> 'a) -> ('a -> 'a)` .

Is your head spinning yet? You might want to read it again until it sinks in.

Quite a sophisticated deduction for one line of code. Luckily the compiler does all this for us. But you will need to understand this kind of thing if you have problems and you have to determine what the compiler is doing.

Let's test it! It's actually much simpler to understand in practice than it is in theory.

```
let doItTwice f  = (f >> f)

let add3 x = x + 3
let add6 = doItTwice add3
// test
add6 5             // result = 11

let square x = x * x
let fourthPower = doItTwice square
// test
fourthPower 3      // result = 81

let chittyBang x = "Chitty " + x + " Bang"
let chittyChittyBangBang = doItTwice chittyBang
// test
chittyChittyBangBang "&"      // result = "Chitty Chitty & Bang Bang"
```

Hopefully, that makes more sense now.

Things that can go wrong with type inference

The type inference isn't perfect, alas. Sometimes the compiler just doesn't have a clue what to do. Again, understanding what is happening will really help you stay calm instead of wanting to kill the compiler. Here are some of the main reasons for type errors:

- Declarations out of order
- Not enough information
- Overloaded methods
- Quirks of generic numeric functions

Declarations out of order

A basic rule is that you must declare functions before they are used.

This code fails:

```
let square2 x = square x   // fails: square not defined
let square x = x * x
```

But this is ok:

```
let square x = x * x
let square2 x = square x   // square already defined earlier
```

And unlike C#, in F# the order of file compilation is important, so do make sure the files are being compiled in the right order. (In Visual Studio, you can change the order from the context menu).

Recursive or simultaneous declarations

A variant of the "out of order" problem occurs with recursive functions or definitions that have to refer to each other. No amount of reordering will help in this case -- we need to use additional keywords to help the compiler.

When a function is being compiled, the function identifier is not available to the body. So if you define a simple recursive function, you will get a compiler error. The fix is to add the "rec" keyword as part of the function definition. For example:

```
// the compiler does not know what "fib" means
let fib n =
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
   // error FS0039: The value or constructor 'fib' is not defined
```

Here's the fixed version with "rec fib" added to indicate it is recursive:

```
let rec fib n =              // LET REC rather than LET
   if n <= 2 then 1
   else fib (n - 1) + fib (n - 2)
```

A similar " `let rec ? and` " syntax is used for two functions that refer to each other. Here is a very contrived example that fails if you do not have the " `rec` " keyword.

```
let rec showPositiveNumber x =               // LET REC rather than LET
   match x with
   | x when x >= 0 -> printfn "%i is positive" x
   | _ -> showNegativeNumber x

and showNegativeNumber x =                   // AND rather than LET

   match x with
   | x when x < 0 -> printfn "%i is negative" x
   | _ -> showPositiveNumber x
```

The " `and` " keyword can also be used to declare simultaneous types in a similar way.

```
type A = None | AUsesB of B
   // error FS0039: The type 'B' is not defined
type B = None | BUsesA of A
```

Fixed version:

```
type A = None | AUsesB of B
and B = None | BUsesA of A    // use AND instead of TYPE
```

Not enough information

Sometimes, the compiler just doesn't have enough information to determine a type. In the following example, the compiler doesn't know what type the `Length` method is supposed to work on. But it can't make it generic either, so it complains.

```
let stringLength s = s.Length
  // error FS0072: Lookup on object of indeterminate type
  // based on information prior to this program point.
  // A type annotation may be needed ...
```

These kinds of error can be fixed with explicit annotations.

```
let stringLength (s:string) = s.Length
```

Occasionally there does appear to be enough information, but still the compiler doesn't seem to recognize it. For example, it's obvious to a human that the `List.map` function (below) is being applied to a list of strings, so why does `x.Length` cause an error?

```
List.map (fun x -> x.Length) ["hello"; "world"]       //not ok
```

The reason is that the F# compiler is currently a one-pass compiler, and so information later in the program is ignored if it hasn't been parsed yet. (The F# team have said that it is possible to make the compiler more sophisticated, but it would work less well with Intellisense and might produce more unfriendly and obscure error messages. So for now, we will have to live with this limitation.)

So in cases like this, you can always explicitly annotate:

```
List.map (fun (x:string) -> x.Length) ["hello"; "world"]       // ok
```

But another, more elegant way that will often fix the problem is to rearrange things so the known types come first, and the compiler can digest them before it moves to the next clause.

```
["hello"; "world"] |> List.map (fun s -> s.Length)   //ok
```

Functional programmers strive to avoid explicit type annotations, so this makes them much happier!

This technique can be used more generally in other areas as well; a rule of thumb is to try to put the things that have "known types" earlier than things that have "unknown types".

Overloaded methods

When calling an external class or method in .NET, you will often get errors due to overloading.

In many cases, such as the concat example below, you will have to explicitly annotate the parameters of the external function so that the compiler knows which overloaded method to call.

```
let concat x = System.String.Concat(x)           //fails
let concat (x:string) = System.String.Concat(x)  //works
let concat x = System.String.Concat(x:string)    //works
```

Sometimes the overloaded methods have different argument names, in which case you can also give the compiler a clue by naming the arguments. Here is an example for the `StreamReader` constructor.

```
let makeStreamReader x = new System.IO.StreamReader(x)        //fails
let makeStreamReader x = new System.IO.StreamReader(path=x)   //works
```

Quirks of generic numeric functions

Numeric functions can be somewhat confusing. There often appear generic, but once they are bound to a particular numeric type, they are fixed, and using them with a different numeric type will cause an error. The following example demonstrates this:

```
let myNumericFn x = x * x
myNumericFn 10
myNumericFn 10.0             //fails
  // error FS0001: This expression was expected to have
  // type int but has type float

let myNumericFn2 x = x * x
myNumericFn2 10.0
myNumericFn2 10               //fails
  // error FS0001: This expression was expected to have
  // type float but has type int
```

There is a way round this for numeric types using the "inline" keyword and "static type parameters". I won't discuss these concepts here, but you can look them up in the F# reference at MSDN.

"Not enough information" troubleshooting summary

So to summarize, the things that you can do if the compiler is complaining about missing types, or not enough information, are:

- Define things before they are used (this includes making sure the files are compiled in the right order)
- Put the things that have "known types" earlier than things that have "unknown types". In particular, you might be able reorder pipes and similar chained functions so that the typed objects come first.
- Annotate as needed. One common trick is to add annotations until everything works, and then take them away one by one until you have the minimum needed. Do try to avoid annotating if possible. Not only is it not aesthetically pleasing, but it makes the code more brittle. It is a lot easier to change types if there are no explicit dependencies on them.

Debugging type inference issues

Once you have ordered and annotated everything, you will probably still get type errors, or find that functions are less generic than expected. With what you have learned so far, you should have the tools to determine why this happened (although it can still be painful).

For example:

```
let myBottomLevelFn x = x

let myMidLevelFn x =
   let y = myBottomLevelFn x
   // some stuff
   let z= y
   // some stuff
   printf "%s" z         // this will kill your generic types!
   // some more stuff
   x

let myTopLevelFn x =
   // some stuff
   myMidLevelFn x
   // some more stuff
   x
```

In this example, we have a chain of functions. The bottom level function is definitely generic, but what about the top level one? Well often, we might expect it be generic but instead it is not. In this case we have:

```
val myTopLevelFn : string -> string
```

What went wrong? The answer is in the midlevel function. The `%s` on z forced it be a string, which forced y and then x to be strings too.

Now this is a pretty obvious example, but with thousands of lines of code, a single line might be buried away that causes an issue. One thing that can help is to look at all the signatures; in this case the signatures are:

```
val myBottomLevelFn : 'a -> 'a       // generic as expected
val myMidLevelFn : string -> string  // here's the clue! Should be generic
val myTopLevelFn : string -> string
```

When you find a signature that is unexpected you know that it is the guilty party. You can then drill down into it and repeat the process until you find the problem.

Choosing between collection functions

There's more to learning a new language than the language itself. In order to be productive, you need to memorize a big chunk of the standard library and be aware of most of the rest of it. For example, if you know C#, you can pick up Java-the-language quite quickly, but you won't really get up to speed until you are comfortable with the Java Class Library as well.

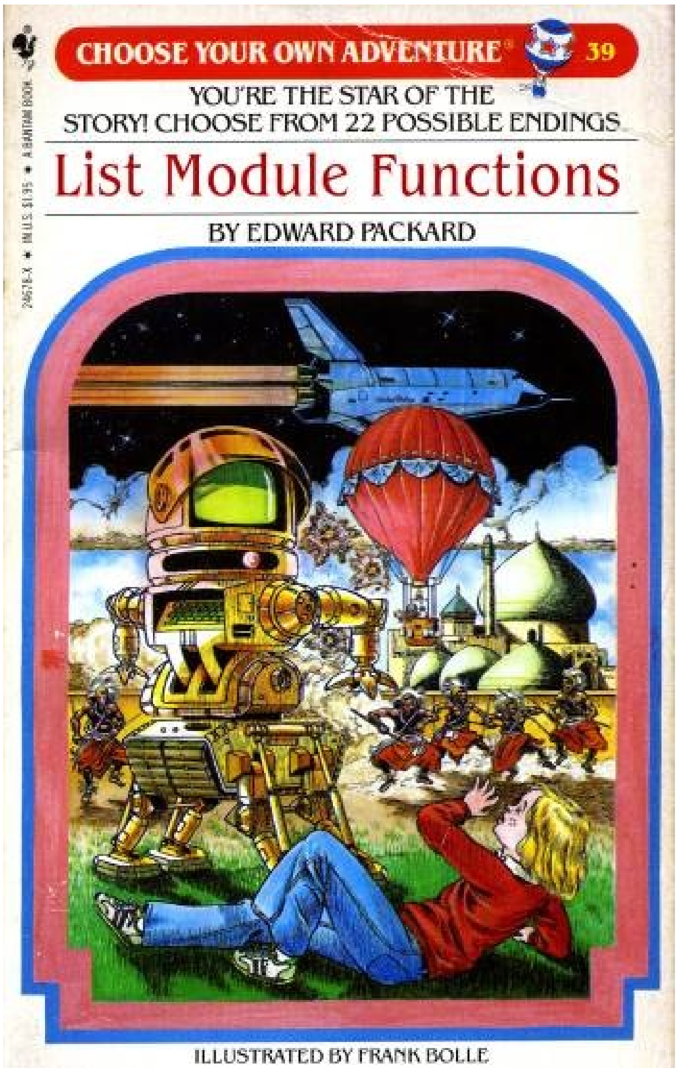
Similarly, you can't really be effective in F# until you have some familiarity with all the F# functions that work with collections.

In C# there are only a few LINQ methods you need to know[1] (`Select` , `Where` , and so on). But in F#, there are currently almost 100 functions in the List module (and similar counts in the Seq and Array modules). That's a lot!

[1] Yes, there are more, but you can get by with just a few. In F# it's more important to know them all.

If you are coming to F# from C#, then, the large number of list functions can be overwhelming.

So I have written this post to help guide you to the one you want. And for fun, I've done it in a "Choose Your Own Adventure" style!



What collection do I want?

First, a table with information about the different kinds of standard collections. There are five "native" F# ones: `list` , `seq` , `array` , `map` and `set` , and `ResizeArray` and `IDictionary` are also often used.

	Immutable?	Notes
list	Yes	**Pros:** • Pattern matching available. • Complex iteration available via recursion. • Forward iteration is fast. Prepending is fast. **Cons:** • Indexed access and other access styles are slow.
seq	Yes	Alias for `IEnumerable`. **Pros:** • Lazy evaluation • Memory efficient (only one element at a time loaded) • Can represent an infinite sequence. • Interop with .NET libraries that use IEnumerable. **Cons:** • No pattern matching. • Forward only iteration. • Indexed access and other access styles are slow.
array	No	Same as BCL `Array`. **Pros:** • Fast random access • Memory efficient and cache locality, especially with structs. • Interop with .NET libraries that use Array. • Support for 2D, 3D and 4D arrays **Cons:** • Limited pattern matching. • Not persistent.
map	Yes	Immutable dictionary. Requires keys to implement `IComparable`.
set	Yes	Immutable set. Requires elements to implement `IComparable`.
ResizeArray	No	Alias for BCL `List`. Pros and cons similar to array, but resizable.
IDictionary	Yes	For an alternate dictionary that does not requires elements to implement `IComparable`, you can use the BCL IDictionary. The constructor is `dict` in F#. Note that mutation methods such as `Add` are present, but will cause a runtime error if called.

These are the main collection types that you will encounter in F#, and will be good enough for all common cases.

If you need other kinds of collections though, there are lots of choices:

- You can use the collection classes in .NET, either the traditional, mutable ones or the newer ones such as those in the System.Collections.Immutable namespace.
- Alternatively, you can use one of the F# collection libraries:
    - **FSharpx.Collections**, part of the FSharpx series of projects.
    - **ExtCore**. Some of these are drop-in (almost) replacements for the Map and Set types in FSharp.Core which provide improved performance in specific scenarios (e.g., HashMap). Others provide unique functionality to help tackle specific coding tasks (e.g., LazyList and LruCache).
    - **Funq**: high performance, immutable data structures for .NET.
    - **Persistent**: some efficient persistent (immutable) data structures.

About the documentation

All functions are available for `list`, `seq` and `array` in F# v4 unless noted. The `Map` and `Set` modules have some of them as well, but I won't be discussing `map` and `set` here.

For the function signatures I will use `list` as the standard collection type. The signatures for the `seq` and `array` versions will be similar.

Many of these functions are not yet documented on MSDN so I'm going to link directly to the source code on GitHub, which has the up-to-date comments. Click on the function name for the link.

Note on availability

The availability of these functions may depend on which version of F# you use.

- In F# version 3 (Visual Studio 2013), there was some degree of inconsistency between Lists, Arrays and Sequences.
- In F# version 4 (Visual Studio 2015), this inconsistency has been eliminated, and almost all functions are available for all three collection types.

If you want to know what changed between F# v3 and F# v4, please see this chart (from here). The chart shows the new APIs in F# v4 (green), previously-existing APIs (blue), and intentional remaining gaps (white).

Some of the functions documented below are not in this chart -- these are newer still! If you are using an older version of F#, you can simply reimplement them yourself using the code on GitHub.

With that disclaimer out of the way, you can start your adventure!

Table of contents

- 1. What kind of collection do you have?
- 2. Creating a new collection
- 3. Creating a new empty or one-element collection
- 4. Creating a new collection of known size
- 5. Creating a new collection of known size with each element having the same value
- 6. Creating a new collection of known size with each element having a different value
- 7. Creating a new infinite collection
- 8. Creating a new collection of indefinite size
- 9. Working with one list
- 10. Getting an element at a known position
- 11. Getting an element by searching
- 12. Getting a subset of elements from a collection
- 13. Partitioning, chunking and grouping
- 14. Aggregating or summarizing a collection
- 15. Changing the order of the elements
- 16. Testing the elements of a collection
- 17. Transforming each element to something different
- 18. Iterating over each element
- 19. Threading state through an iteration
- 20. Working with the index of each element
- 21. Transforming the whole collection to a different collection type
- 22. Changing the behavior of the collection as a whole
- 23. Working with two collections
- 24. Working with three collections
- 25. Working with more than three collections
- 26. Combining and uncombining collections
- 27. Other array-only functions
- 28. Using sequences with disposables

---

1. What kind of collection do you have?

What kind of collection do you have?

- If you don't have a collection, and want to create one, go to section 2.
- If you already have a collection that you want to work with, go to section 9.
- If you have two collections that you want to work with, go to section 23.
- If you have three collections that you want to work with, go to section 24.
- If you have more than three collections that you want to work with, go to section 25.
- If you want to combine or uncombine collections, go to section 26.

---

2. Creating a new collection

So you want to create a new collection. How do you want to create it?

- If the new collection will be empty or will have one element, go to section 3.
- If the new collection is a known size, go to section 4.
- If the new collection is potentially infinite, go to section 7.
- If you don't know how big the collection will be, go to section 8.

---

3. Creating a new empty or one-element collection

If you want to create a new empty or one-element collection, use these functions:

- `empty : 'T list`. Returns an empty list of the given type.
- `singleton : value:'T -> 'T list`. Returns a list that contains one item only.

If you know the size of the collection in advance, it is generally more efficient to use a different function. See section 4 below.

Usage examples

```
let list0 = List.empty
// list0 = []

let list1 = List.singleton "hello"
// list1 = ["hello"]
```

4. Creating a new collection of known size

- If all elements of the collection will have the same value, go to section 5.
- If elements of the collection could be different, go to section 6.

5. Creating a new collection of known size with each element having the same value

If you want to create a new collection of known size with each element having the same value, you want to use `replicate`:

- `replicate : count:int -> initial:'T -> 'T list`. Creates a collection by replicating the given initial value.
- (Array only) `create : count:int -> value:'T -> 'T[]`. Creates an array whose elements are all initially the supplied value.
- (Array only) `zeroCreate : count:int -> 'T[]`. Creates an array where the entries are initially the default value.

`Array.create` is basically the same as `replicate` (although with a subtly different implementation!) but `replicate` was only implemented for `Array` in F# v4.

Usage examples

```
let repl = List.replicate 3 "hello"
// val repl : string list = ["hello"; "hello"; "hello"]

let arrCreate = Array.create 3 "hello"
// val arrCreate : string [] = [|"hello"; "hello"; "hello"|]

let intArr0 : int[] = Array.zeroCreate 3
// val intArr0 : int [] = [|0; 0; 0|]

let stringArr0 : string[] = Array.zeroCreate 3
// val stringArr0 : string [] = [|null; null; null|]
```

Note that for `zeroCreate`, the target type must be known to the compiler.

6. Creating a new collection of known size with each element having a different value

If you want to create a new collection of known size with each element having a potentially different value, you can choose one of three ways:

- `init : length:int -> initializer:(int -> 'T) -> 'T list`. Creates a collection by calling the given generator on each index.
- For lists and arrays, you can also use the literal syntax such as `[1; 2; 3]` (lists) and `[|1; 2; 3|]` (arrays).
- For lists and arrays and seqs, you can use the comprehension syntax `for .. in .. do .. yield`.

Usage examples

```
// using list initializer
let listInit1 = List.init 5 (fun i-> i*i)
// val listInit1 : int list = [0; 1; 4; 9; 16]

// using list comprehension
let listInit2 = [for i in [1..5] do yield i*i]
// val listInit2 : int list = [1; 4; 9; 16; 25]

// literal
let listInit3 = [1; 4; 9; 16; 25]
// val listInit3 : int list = [1; 4; 9; 16; 25]

let arrayInit3 = [|1; 4; 9; 16; 25|]
// val arrayInit3 : int [] = [|1; 4; 9; 16; 25|]
```

Literal syntax allows for an increment as well:

```
// literal with +2 increment
let listOdd= [1..2..10]
// val listOdd : int list = [1; 3; 5; 7; 9]
```

The comprehension syntax is even more flexible because you can `yield` more than once:

```
// using list comprehension
let listFunny = [
    for i in [2..3] do
        yield i
        yield i*i
        yield i*i*i
        ]
// val listFunny : int list = [2; 4; 8; 3; 9; 27]
```

and it can also be used as a quick and dirty inline filter:

```
let primesUpTo n =
   let rec sieve l  =
      match l with
      | [] -> []
      | p::xs ->
            p :: sieve [for x in xs do if (x % p) > 0 then yield x]
   [2..n] |> sieve

primesUpTo 20
// [2; 3; 5; 7; 11; 13; 17; 19]
```

Two other tricks:

- You can use `yield!` to return a list rather than a single value
- You can also use recursion

Here is an example of both tricks being used to count up to 10 by twos:

```
let rec listCounter n = [
    if n <= 10 then
        yield n
        yield! listCounter (n+2)
    ]

listCounter 3
// val it : int list = [3; 5; 7; 9]
listCounter 4
// val it : int list = [4; 6; 8; 10]
```

---

7. Creating a new infinite collection

If you want an infinite list, you have to use a seq rather than a list or array.

- `initInfinite : initializer:(int -> 'T) -> seq<'T>`. Generates a new sequence which,

when iterated, will return successive elements by calling the given function.
- You can also use a seq comprehension with a recursive loop to generate an infinite sequence.

Usage examples

```
// generator version
let seqOfSquares = Seq.initInfinite (fun i -> i*i)
let firstTenSquares = seqOfSquares |> Seq.take 10

firstTenSquares |> List.ofSeq // [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]

// recursive version
let seqOfSquares_v2 =
    let rec loop n = seq {
        yield n * n
        yield! loop (n+1)
        }
    loop 1
let firstTenSquares_v2 = seqOfSquares_v2 |> Seq.take 10
```

8. Creating a new collection of indefinite size

Sometimes you don't know how big the collection will be in advance. In this case you need a function that will keep adding elements until it gets a signal to stop. `unfold` is your friend here, and the "signal to stop" is whether you return a `None` (stop) or a `Some` (keep going).

- `unfold : generator:('State -> ('T * 'State) option) -> state:'State -> 'T list`. Returns a collection that contains the elements generated by the given computation.

Usage examples

This example reads from the console in a loop until an empty line is entered:

```
let getInputFromConsole lineNo =
    let text = System.Console.ReadLine()
    if System.String.IsNullOrEmpty(text) then
        None
    else
        // return value and new threaded state
        // "text" will be in the generated sequence
        Some (text,lineNo+1)

let listUnfold = List.unfold getInputFromConsole 1
```

`unfold` requires that a state be threaded through the generator. You can ignore it (as in the `ReadLine` example above), or you can use it to keep track of what you have done so far. For example, you can create a Fibonacci series generator using `unfold`:

```
let fibonacciUnfolder max (f1,f2)  =
    if f1 > max then
        None
    else
        // return value and new threaded state
        let fNext = f1 + f2
        let newState = (f2,fNext)
        // f1 will be in the generated sequence
        Some (f1,newState)

let fibonacci max = List.unfold (fibonacciUnfolder max) (1,1)
fibonacci 100
// int list = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

9. Working with one list

If you are working with one list and...

- If you want to get an element at a known position, go to section 10
- If you want to get one element by searching, go to section 11
- If you want to get a subset of the collection, go to section 12
- If you want to partition, chunk, or group a collection into smaller collections, go to section 13
- If you want to aggregate or summarize the collection into a single value, go to section 14
- If you want to change the order of the elements, go to section 15
- If you want to test the elements in the collection, go to section 16
- If you want to transform each element to something different, go to section 17

- If you want to iterate over each element, go to section 18
- If you want to thread state through an iteration, go to section 19
- If you need to know the index of each element while you are iterating or mapping, go to section 20
- If you want to transform the whole collection to a different collection type, go to section 21
- If you want to change the behaviour of the collection as a whole, go to section 22
- If you want to mutate the collection in place, go to section 27
- If you want to use a lazy collection with an IDisposable, go to section 28

---

10. Getting an element at a known position

The following functions get a element in the collection by position:

- `head : list:'T list -> 'T`. Returns the first element of the collection.
- `last : list:'T list -> 'T`. Returns the last element of the collection.
- `item : index:int -> list:'T list -> 'T`. Indexes into the collection. The first element has index 0.
  NOTE: Avoid using `nth` and `item` for lists and sequences. They are not designed for random access, and so they will be slow in general.
- `nth : list:'T list -> index:int -> 'T`. The older version of `item`. NOTE: Deprecated in v4 -- use `item` instead.
- (Array only) `get : array:'T[] -> index:int -> 'T`. Yet another version of `item`.
- `exactlyOne : list:'T list -> 'T`. Returns the only element of the collection.

But what if the collection is empty? Then `head` and `last` will fail with an exception (ArgumentException).

And if the index is not found in the collection? Then another exception again (ArgumentException for lists, IndexOutOfRangeException for arrays).

I would therefore recommend that you avoid these functions in general and use the `tryXXX` equivalents below:

- `tryHead : list:'T list -> 'T option`. Returns the first element of the collection, or None if the collection is empty.
- `tryLast : list:'T list -> 'T option`. Returns the last element of the collection, or None if the collection is empty.
- `tryItem : index:int -> list:'T list -> 'T option`. Indexes into the collection, or None if the index is not valid.

Usage examples

```
let head = [1;2;3] |> List.head
// val head : int = 1

let badHead : int = [] |> List.head
// System.ArgumentException: The input list was empty.

let goodHeadOpt =
    [1;2;3] |> List.tryHead
// val goodHeadOpt : int option = Some 1

let badHeadOpt : int option =
    [] |> List.tryHead
// val badHeadOpt : int option = None

let goodItemOpt =
    [1;2;3] |> List.tryItem 2
// val goodItemOpt : int option = Some 3

let badItemOpt =
    [1;2;3] |> List.tryItem 99
// val badItemOpt : int option = None
```

As noted, the `item` function should be avoided for lists. For example, if you want to process each item in a list, and you come from an imperative background, you might write a loop with something like this:

```
// Don't do this!
let helloBad =
    let list = ["a";"b";"c"]
    let listSize = List.length list
    [ for i in [0..listSize-1] do
        let element = list |> List.item i
        yield "hello " + element
    ]
// val helloBad : string list = ["hello a"; "hello b"; "hello c"]
```

Don't do that! Use something like `map` instead. It's both more concise and more efficient:

```
let helloGood =
    let list = ["a";"b";"c"]
    list |> List.map (fun element -> "hello " + element)
// val helloGood : string list = ["hello a"; "hello b"; "hello c"]
```

11. Getting an element by searching

You can search for an element or its index using `find` and `findIndex`:

- `find : predicate:('T -> bool) -> list:'T list -> 'T`. Returns the first element for which the given function returns true.
- `findIndex : predicate:('T -> bool) -> list:'T list -> int`. Returns the index of the first element for which the given function returns true.

And you can also search backwards:

- `findBack : predicate:('T -> bool) -> list:'T list -> 'T`. Returns the last element for which the given function returns true.
- `findIndexBack : predicate:('T -> bool) -> list:'T list -> int`. Returns the index of the last element for which the given function returns true.

But what if the item cannot be found? Then these will fail with an exception (`KeyNotFoundException`).

I would therefore recommend that, as with `head` and `item`, you avoid these functions in general and use the `tryXXX` equivalents below:

- `tryFind : predicate:('T -> bool) -> list:'T list -> 'T option`. Returns the first element for which the given function returns true, or None if no such element exists.
- `tryFindBack : predicate:('T -> bool) -> list:'T list -> 'T option`. Returns the last element for which the given function returns true, or None if no such element exists.
- `tryFindIndex : predicate:('T -> bool) -> list:'T list -> int option`. Returns the index of the first element for which the given function returns true, or None if no such element exists.
- `tryFindIndexBack : predicate:('T -> bool) -> list:'T list -> int option`. Returns the index of the last element for which the given function returns true, or None if no such element exists.

If you are doing a `map` before a `find` you can often combine the two steps into a single one using `pick` (or better, `tryPick`). See below for a usage example.

- `pick : chooser:('T -> 'U option) -> list:'T list -> 'U`. Applies the given function to successive elements, returning the first result where the chooser function returns Some.
- `tryPick : chooser:('T -> 'U option) -> list:'T list -> 'U option`. Applies the given function to successive elements, returning the first result where the chooser function returns Some, or None if no such element exists.

Usage examples

```
let listOfTuples = [ (1,"a"); (2,"b"); (3,"b"); (4,"a"); ]

listOfTuples |> List.find ( fun (x,y) -> y = "b")
// (2, "b")

listOfTuples |> List.findBack ( fun (x,y) -> y = "b")
// (3, "b")

listOfTuples |> List.findIndex ( fun (x,y) -> y = "b")
// 1

listOfTuples |> List.findIndexBack ( fun (x,y) -> y = "b")
// 2

listOfTuples |> List.find ( fun (x,y) -> y = "c")
// KeyNotFoundException
```

With `pick`, rather than returning a bool, you return an option:

```
listOfTuples |> List.pick ( fun (x,y) -> if y = "b" then Some (x,y) else None)
// (2, "b")
```

Pick vs. Find

That 'pick' function might seem unnecessary, but it is useful when dealing with functions that return options.

For example, say that there is a function `tryInt` that parses a string and returns `Some int` if the string is a valid int, otherwise `None`.

```
// string -> int option
let tryInt str =
    match System.Int32.TryParse(str) with
    | true, i -> Some i
    | false, _ -> None
```

And now say that we want to find the first valid int in a list. The crude way would be:

- map the list using `tryInt`
- find the first one that is a `Some` using `find`
- get the value from inside the option using `Option.get`

The code might look something like this:

```
let firstValidNumber =
    ["a";"2";"three"]
    // map the input
    |> List.map tryInt
    // find the first Some
    |> List.find (fun opt -> opt.IsSome)
    // get the data from the option
    |> Option.get
// val firstValidNumber : int = 2
```

But `pick` will do all these steps at once! So the code becomes much simpler:

```
let firstValidNumber =
    ["a";"2";"three"]
    |> List.pick tryInt
```

If you want to return many elements in the same way as `pick`, consider using `choose` (see section 12).

---

12. Getting a subset of elements from a collection

The previous section was about getting one element. How can you get more than one element? Well you're in luck! There's lots of functions to choose from.

To extract elements from the front, use one of these:

- `take: count:int -> list:'T list -> 'T list`. Returns the first N elements of the collection.
- `takeWhile: predicate:('T -> bool) -> list:'T list -> 'T list`. Returns a collection that contains all elements of the original collection while the given predicate returns true, and then returns no further elements.
- `truncate: count:int -> list:'T list -> 'T list`. Returns at most N elements in a new collection.

To extract elements from the rear, use one of these:

- `skip: count:int -> list: 'T list -> 'T list`. Returns the collection after removing the first N elements.
- `skipWhile: predicate:('T -> bool) -> list:'T list -> 'T list`. Bypasses elements in a collection while the given predicate returns true, and then returns the remaining

elements of the collection.
- `tail: list:'T list -> 'T list`. Returns the collection after removing the first element.

To extract other subsets of elements, use one of these:

- `filter: predicate:('T -> bool) -> list:'T list -> 'T list`. Returns a new collection containing only the elements of the collection for which the given function returns true.
- `except: itemsToExclude:seq<'T> -> list:'T list -> 'T list when 'T : equality`. Returns a new collection with the distinct elements of the input collection which do not appear in the itemsToExclude sequence, using generic hash and equality comparisons to compare values.
- `choose: chooser:('T -> 'U option) -> list:'T list -> 'U list`. Applies the given function to each element of the collection. Returns a collection comprised of the elements where the function returns Some.
- `where: predicate:('T -> bool) -> list:'T list -> 'T list`. Returns a new collection containing only the elements of the collection for which the given predicate returns true. NOTE: "where" is a synonym for "filter".
- (Array only) `sub : 'T [] -> int -> int -> 'T []`. Creates an array that contains the supplied subrange, which is specified by starting index and length.
- You can also use slice syntax: `myArray.[2..5]`. See below for examples.

To reduce the list to distinct elements, use one of these:

- `distinct: list:'T list -> 'T list when 'T : equality`. Returns a collection that contains no duplicate entries according to generic hash and equality comparisons on the entries.
- `distinctBy: projection:('T -> 'Key) -> list:'T list -> 'T list when 'Key : equality`. Returns a collection that contains no duplicate entries according to the generic hash and equality comparisons on the keys returned by the given key-generating function.

Usage examples

Taking elements from the front:

```
[1..10] |> List.take 3
// [1; 2; 3]

[1..10] |> List.takeWhile (fun i -> i < 3)
// [1; 2]

[1..10] |> List.truncate 4
// [1; 2; 3; 4]

[1..2] |> List.take 3
// System.InvalidOperationException: The input sequence has an insufficient number of
elements.

[1..2] |> List.takeWhile (fun i -> i < 3)
// [1; 2]

[1..2] |> List.truncate 4
// [1; 2]   // no error!
```

Taking elements from the rear:

```
[1..10] |> List.skip 3
// [4; 5; 6; 7; 8; 9; 10]

[1..10] |> List.skipWhile (fun i -> i < 3)
// [3; 4; 5; 6; 7; 8; 9; 10]

[1..10] |> List.tail
// [2; 3; 4; 5; 6; 7; 8; 9; 10]

[1..2] |> List.skip 3
// System.ArgumentException: The index is outside the legal range.

[1..2] |> List.skipWhile (fun i -> i < 3)
// []

[1] |> List.tail |> List.tail
// System.ArgumentException: The input list was empty.
```

To extract other subsets of elements:

```
[1..10] |> List.filter (fun i -> i%2 = 0) // even
// [2; 4; 6; 8; 10]

[1..10] |> List.where (fun i -> i%2 = 0) // even
// [2; 4; 6; 8; 10]

[1..10] |> List.except [3;4;5]
// [1; 2; 6; 7; 8; 9; 10]
```

To extract a slice:

```
Array.sub [|1..10|] 3 5
// [|4; 5; 6; 7; 8|]

[1..10].[3..5]
// [4; 5; 6]

[1..10].[3..]
// [4; 5; 6; 7; 8; 9; 10]

[1..10].[..5]
// [1; 2; 3; 4; 5; 6]
```

Note that slicing on lists can be slow, because they are not random access. Slicing on arrays is fast however.

To extract the distinct elements:

```
[1;1;1;2;3;3] |> List.distinct
// [1; 2; 3]

[ (1,"a"); (1,"b"); (1,"c"); (2,"d")] |> List.distinctBy fst
// [(1, "a"); (2, "d")]
```

Choose vs. Filter

As with `pick`, the `choose` function might seem awkward, but it is useful when dealing with functions that return options.

In fact, `choose` is to `filter` as `pick` is to `find`, Rather than using a boolean filter, the signal is `Some` vs. `None`.

As before, say that there is a function `tryInt` that parses a string and returns `Some int` if the string is a valid int, otherwise `None`.

```
// string -> int option
let tryInt str =
    match System.Int32.TryParse(str) with
    | true, i -> Some i
    | false, _ -> None
```

And now say that we want to find all the valid ints in a list. The crude way would be:

- map the list using `tryInt`
- filter to only include the ones that are `Some`

- get the value from inside each option using `Option.get`

The code might look something like this:

```
let allValidNumbers =
    ["a";"2";"three"; "4"]
    // map the input
    |> List.map tryInt
    // include only the "Some"
    |> List.filter (fun opt -> opt.IsSome)
    // get the data from each option
    |> List.map Option.get
// val allValidNumbers : int list = [2; 4]
```

But `choose` will do all these steps at once! So the code becomes much simpler:

```
let allValidNumbers =
    ["a";"2";"three"; "4"]
    |> List.choose tryInt
```

If you already have a list of options, you can filter and return the "Some" in one step by passing `id` into `choose`:

```
let reduceOptions =
    [None; Some 1; None; Some 2]
    |> List.choose id
// val reduceOptions : int list = [1; 2]
```

If you want to return the first element in the same way as `choose`, consider using `pick` (see section 11).

If you want to do a similar action as `choose` but for other wrapper types (such as a Success/Failure result), there is a discussion here.

13. Partitioning, chunking and grouping

There are lots of different ways to split a collection! Have a look at the usage examples to see the differences:

- `chunkBySize: chunkSize:int -> list:'T list -> 'T list list`. Divides the input collection into chunks of size at most `chunkSize`.
- `groupBy : projection:('T -> 'Key) -> list:'T list -> ('Key * 'T list) list when 'Key`

`: equality`. Applies a key-generating function to each element of a collection and yields a list of unique keys. Each unique key contains a list of all elements that match to this key.

- `pairwise: list:'T list -> ('T * 'T) list`. Returns a collection of each element in the input collection and its predecessor, with the exception of the first element which is only returned as the predecessor of the second element.
- (Except Seq) `partition: predicate:('T -> bool) -> list:'T list -> ('T list * 'T list)`. Splits the collection into two collections, containing the elements for which the given predicate returns true and false respectively.
- (Except Seq) `splitAt: index:int -> list:'T list -> ('T list * 'T list)`. Splits a collection into two collections at the given index.
- `splitInto: count:int -> list:'T list -> 'T list list`. Splits the input collection into at most count chunks.
- `windowed : windowSize:int -> list:'T list -> 'T list list`. Returns a list of sliding windows containing elements drawn from the input collection. Each window is returned as a fresh collection. Unlike `pairwise` the windows are collections, not tuples.

Usage examples

```
[1..10] |> List.chunkBySize 3
// [[1; 2; 3]; [4; 5; 6]; [7; 8; 9]; [10]]
// note that the last chunk has one element

[1..10] |> List.splitInto 3
// [[1; 2; 3; 4]; [5; 6; 7]; [8; 9; 10]]
// note that the first chunk has four elements

['a'..'i'] |> List.splitAt 3
// (['a'; 'b'; 'c'], ['d'; 'e'; 'f'; 'g'; 'h'; 'i'])

['a'..'e'] |> List.pairwise
// [('a', 'b'); ('b', 'c'); ('c', 'd'); ('d', 'e')]

['a'..'e'] |> List.windowed 3
// [['a'; 'b'; 'c']; ['b'; 'c'; 'd']; ['c'; 'd'; 'e']]

let isEven i = (i%2 = 0)
[1..10] |> List.partition isEven
// ([2; 4; 6; 8; 10], [1; 3; 5; 7; 9])

let firstLetter (str:string) = str.[0]
["apple"; "alice"; "bob"; "carrot"] |> List.groupBy firstLetter
// [('a', ["apple"; "alice"]); ('b', ["bob"]); ('c', ["carrot"])]
```

All the functions other than `splitAt` and `pairwise` handle edge cases gracefully:

```
[1] |> List.chunkBySize 3
// [[1]]

[1] |> List.splitInto 3
// [[1]]

['a'; 'b'] |> List.splitAt 3
// InvalidOperationException: The input sequence has an insufficient number of element
s.

['a'] |> List.pairwise
// InvalidOperationException: The input sequence has an insufficient number of element
s.

['a'] |> List.windowed 3
// []

[1] |> List.partition isEven
// ([], [1])

[] |> List.groupBy firstLetter
//  []
```

14. Aggregating or summarizing a collection

The most generic way to aggregate the elements in a collection is to use `reduce` :

- `reduce : reduction:('T -> 'T -> 'T) -> list:'T list -> 'T`. Apply a function to each element of the collection, threading an accumulator argument through the computation.
- `reduceBack : reduction:('T -> 'T -> 'T) -> list:'T list -> 'T`. Applies a function to each element of the collection, starting from the end, threading an accumulator argument through the computation.

and there are specific versions of `reduce` for frequently used aggregations:

- `max : list:'T list -> 'T when 'T : comparison`. Return the greatest of all elements of the collection, compared via Operators.max.
- `maxBy : projection:('T -> 'U) -> list:'T list -> 'T when 'U : comparison`. Returns the greatest of all elements of the collection, compared via Operators.max on the function result.
- `min : list:'T list -> 'T when 'T : comparison`. Returns the lowest of all elements of the collection, compared via Operators.min.
- `minBy : projection:('T -> 'U) -> list:'T list -> 'T when 'U : comparison`. Returns the lowest of all elements of the collection, compared via Operators.min on the function

result.

- `sum : list:'T list -> 'T when 'T has static members (+) and Zero`. Returns the sum of the elements in the collection.
- `sumBy : projection:('T -> 'U) -> list:'T list -> 'U when 'U has static members (+) and Zero`. Returns the sum of the results generated by applying the function to each element of the collection.
- `average : list:'T list -> 'T when 'T has static members (+) and Zero and DivideByInt`. Returns the average of the elements in the collection. Note that a list of ints cannot be averaged -- they must be cast to floats or decimals.
- `averageBy : projection:('T -> 'U) -> list:'T list -> 'U when 'U has static members (+) and Zero and DivideByInt`. Returns the average of the results generated by applying the function to each element of the collection.

Finally there are some counting functions:

- `length: list:'T list -> int`. Returns the length of the collection.
- `countBy : projection:('T -> 'Key) -> list:'T list -> ('Key * int) list when 'Key : equality`. Applies a key-generating function to each element and returns a collection yielding unique keys and their number of occurrences in the original collection.

Usage examples

`reduce` is a variant of `fold` without an initial state -- see section 19 for more on `fold`. One way to think of it is just inserting a operator between each element.

```
["a";"b";"c"] |> List.reduce (+)
// "abc"
```

is the same as

```
"a" + "b" + "c"
```

Here's another example:

```
[2;3;4] |> List.reduce (*)
// is same as
2 * 3 * 4
// Result is 24
```

Some ways of combining elements depend on the order of combining, and so there are two variants of "reduce":

- `reduce` moves forward through the list.
- `reduceBack`, not surprisingly, moves backwards through the list.

Here's a demonstration of the difference. First `reduce`:

```
[1;2;3;4] |> List.reduce (fun state x -> (state)*10 + x)

// built up from                // state at each step
1                               // 1
(1)*10 + 2                      // 12
((1)*10 + 2)*10 + 3             // 123
(((1)*10 + 2)*10 + 3)*10 + 4    // 1234

// Final result is 1234
```

Using the *same* combining function with `reduceBack` produces a different result! It looks like this:

```
[1;2;3;4] |> List.reduceBack (fun x state -> x + 10*(state))

// built up from                // state at each step
4                               // 4
3 + 10*(4)                      // 43
2 + 10*(3 + 10*(4))             // 432
1 + 10*(2 + 10*(3 + 10*(4)))    // 4321

// Final result is 4321
```

Again, see section 19 for a more detailed discussion of the related functions `fold` and `foldBack`.

The other aggregation functions are much more straightforward.

```
type Suit = Club | Diamond | Spade | Heart
type Rank = Two | Three | King | Ace
let cards = [ (Club,King); (Diamond,Ace); (Spade,Two); (Heart,Three); ]

cards |> List.max        // (Heart, Three)
cards |> List.maxBy snd  // (Diamond, Ace)
cards |> List.min        // (Club, King)
cards |> List.minBy snd  // (Spade, Two)

[1..10] |> List.sum
// 55

[ (1,"a"); (2,"b") ] |> List.sumBy fst
// 3

[1..10] |> List.average
// The type 'int' does not support the operator 'DivideByInt'

[1..10] |> List.averageBy float
// 5.5

[ (1,"a"); (2,"b") ] |> List.averageBy (fst >> float)
// 1.5

[1..10] |> List.length
// 10

[ ("a","A"); ("b","B"); ("a","C") ]  |> List.countBy fst
// [("a", 2); ("b", 1)]

[ ("a","A"); ("b","B"); ("a","C") ]  |> List.countBy snd
// [("A", 1); ("B", 1); ("C", 1)]
```

Most of the aggregation functions do not like empty lists! You might consider using one of the `fold` functions to be safe -- see section 19.

```
let emptyListOfInts : int list = []

emptyListOfInts |> List.reduce (+)
// ArgumentException: The input list was empty.

emptyListOfInts |> List.max
// ArgumentException: The input sequence was empty.

emptyListOfInts |> List.min
// ArgumentException: The input sequence was empty.

emptyListOfInts |> List.sum
// 0

emptyListOfInts |> List.averageBy float
// ArgumentException: The input sequence was empty.

let emptyListOfTuples : (int*int) list = []
emptyListOfTuples |> List.countBy fst
// (int * int) list = []
```

---

15. Changing the order of the elements

You can change the order of the elements using reversing, sorting and permuting. All of the following return *new* collections:

- `rev: list:'T list -> 'T list`. Returns a new collection with the elements in reverse order.
- `sort: list:'T list -> 'T list when 'T : comparison`. Sorts the given collection using Operators.compare.
- `sortDescending: list:'T list -> 'T list when 'T : comparison`. Sorts the given collection in descending order using Operators.compare.
- `sortBy: projection:('T -> 'Key) -> list:'T list -> 'T list when 'Key : comparison`. Sorts the given collection using keys given by the given projection. Keys are compared using Operators.compare.
- `sortByDescending: projection:('T -> 'Key) -> list:'T list -> 'T list when 'Key : comparison`. Sorts the given collection in descending order using keys given by the given projection. Keys are compared using Operators.compare.
- `sortWith: comparer:('T -> 'T -> int) -> list:'T list -> 'T list`. Sorts the given collection using the given comparison function.
- `permute : indexMap:(int -> int) -> list:'T list -> 'T list`. Returns a collection with all elements permuted according to the specified permutation.

And there are also some array-only functions that sort in place:

- (Array only) `sortInPlace: array:'T[] -> unit when 'T : comparison`. Sorts the elements of an array by mutating the array in-place. Elements are compared using Operators.compare.
- (Array only) `sortInPlaceBy: projection:('T -> 'Key) -> array:'T[] -> unit when 'Key : comparison`. Sorts the elements of an array by mutating the array in-place, using the given projection for the keys. Keys are compared using Operators.compare.
- (Array only) `sortInPlaceWith: comparer:('T -> 'T -> int) -> array:'T[] -> unit`. Sorts the elements of an array by mutating the array in-place, using the given comparison function as the order.

Usage examples

```
[1..5] |> List.rev
// [5; 4; 3; 2; 1]

[2;4;1;3;5] |> List.sort
// [1; 2; 3; 4; 5]

[2;4;1;3;5] |> List.sortDescending
// [5; 4; 3; 2; 1]

[ ("b","2"); ("a","3"); ("c","1") ]  |> List.sortBy fst
// [("a", "3"); ("b", "2"); ("c", "1")]

[ ("b","2"); ("a","3"); ("c","1") ]  |> List.sortBy snd
// [("c", "1"); ("b", "2"); ("a", "3")]

// example of a comparer
let tupleComparer tuple1 tuple2  =
    if tuple1 < tuple2 then
        -1
    elif tuple1 > tuple2 then
        1
    else
        0

[ ("b","2"); ("a","3"); ("c","1") ]  |> List.sortWith tupleComparer
// [("a", "3"); ("b", "2"); ("c", "1")]

[1..10] |> List.permute (fun i -> (i + 3) % 10)
// [8; 9; 10; 1; 2; 3; 4; 5; 6; 7]

[1..10] |> List.permute (fun i -> 9 - i)
// [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
```

16. Testing the elements of a collection

These set of functions all return true or false.

- `contains: value:'T -> source:'T list -> bool when 'T : equality`. Tests if the collection contains the specified element.
- `exists: predicate:('T -> bool) -> list:'T list -> bool`. Tests if any element of the collection satisfies the given predicate.
- `forall: predicate:('T -> bool) -> list:'T list -> bool`. Tests if all elements of the collection satisfy the given predicate.
- `isEmpty: list:'T list -> bool`. Returns true if the collection contains no elements, false otherwise.

Usage examples

```
[1..10] |> List.contains 5
// true

[1..10] |> List.contains 42
// false

[1..10] |> List.exists (fun i -> i > 3 && i < 5)
// true

[1..10] |> List.exists (fun i -> i > 5 && i < 3)
// false

[1..10] |> List.forall (fun i -> i > 0)
// true

[1..10] |> List.forall (fun i -> i > 5)
// false

[1..10] |> List.isEmpty
// false
```

17. Transforming each element to something different

I sometimes like to think of functional programming as "transformation-oriented programming", and `map` (aka `Select` in LINQ) is one of the most fundamental ingredients for this approach. In fact, I have devoted a whole series to it here.

- `map: mapping:('T -> 'U) -> list:'T list -> 'U list`. Builds a new collection whose elements are the results of applying the given function to each of the elements of the collection.

Sometimes each element maps to a list, and you want to flatten out all the lists. For this case, use `collect` (aka `SelectMany` in LINQ).

- `collect: mapping:('T -> 'U list) -> list:'T list -> 'U list`. For each element of the list, applies the given function. Concatenates all the results and return the combined list.

Other transformation functions include:

- `choose` in section 12 is a map and option filter combined.
- (Seq only) `cast: source:IEnumerable -> seq<'T>`. Wraps a loosely-typed `System.Collections` sequence as a typed sequence.

Usage examples

Here are some examples of using `map` in the conventional way, as a function that accepts a list and a mapping function and returns a new transformed list:

```
let add1 x = x + 1

// map as a list transformer
[1..5] |> List.map add1
// [2; 3; 4; 5; 6]

// the list being mapped over can contain anything!
let times2 x = x * 2
[ add1; times2] |> List.map (fun f -> f 5)
// [6; 10]
```

You can also think of `map` as a *function transformer*. It turns an element-to-element function into a list-to-list function.

```
let add1ToEachElement = List.map add1
// "add1ToEachElement" transforms lists to lists rather than ints to ints
// val add1ToEachElement : (int list -> int list)

// now use it
[1..5] |> add1ToEachElement
// [2; 3; 4; 5; 6]
```

`collect` works to flatten lists. If you already have a list of lists, you can use `collect` with `id` to flatten them.

```
[2..5] |> List.collect (fun x -> [x; x*x; x*x*x] )
// [2; 4; 8; 3; 9; 27; 4; 16; 64; 5; 25; 125]

// using "id" with collect
let list1 = [1..3]
let list2 = [4..6]
[list1; list2] |> List.collect id
// [1; 2; 3; 4; 5; 6]
```

Seq.cast

Finally, `Seq.cast` is useful when working with older parts of the BCL that have specialized collection classes rather than generics.

For example, the Regex library has this problem, and so the code below won't compile because `MatchCollection` is not an `IEnumerable<T>`

```
open System.Text.RegularExpressions

let matches =
    let pattern = "\d\d\d"
    let matchCollection = Regex.Matches("123 456 789",pattern)
    matchCollection
    |> Seq.map (fun m -> m.Value)     // ERROR
    // ERROR: The type 'MatchCollection' is not compatible with the type 'seq<'a>'
    |> Seq.toList
```

The fix is to cast `MatchCollection` to a `Seq<Match>` and then the code will work nicely:

```
let matches =
    let pattern = "\d\d\d"
    let matchCollection = Regex.Matches("123 456 789",pattern)
    matchCollection
    |> Seq.cast<Match>
    |> Seq.map (fun m -> m.Value)
    |> Seq.toList
// output = ["123"; "456"; "789"]
```

---

18. Iterating over each element

Normally, when processing a collection, we transform each element to a new value using `map`. But occasionally we need to process all the elements with a function which *doesn't* produce a useful value (a "unit function").

- `iter: action:('T -> unit) -> list:'T list -> unit`. Applies the given function to each element of the collection.
- Alternatively, you can use a for-loop. The expression inside a for-loop *must* return `unit`.

Usage examples

The most common examples of unit functions are all about side-effects: printing to the console, updating a database, putting a message on a queue, etc. For the examples below, I will just use `printfn` as my unit function.

```
[1..3] |> List.iter (fun i -> printfn "i is %i" i)
(*
i is 1
i is 2
i is 3
*)

// or using partial application
[1..3] |> List.iter (printfn "i is %i")

// or using a for loop
for i = 1 to 3 do
    printfn "i is %i" i

// or using a for-in loop
for i in [1..3] do
    printfn "i is %i" i
```

As noted above, the expression inside an `iter` or for-loop must return unit. In the following examples, we try to add 1 to the element, and get a compiler error:

```
[1..3] |> List.iter (fun i -> i + 1)
//                               ~~~
// ERROR error FS0001: The type 'unit' does not match the type 'int'

// a for-loop expression *must* return unit
for i in [1..3] do
     i + 1  // ERROR
     // This expression should have type 'unit',
     // but has type 'int'. Use 'ignore' ...
```

If you are sure that this is not a logic bug in your code, and you want to get rid of this error, you can pipe the results into `ignore`:

```
[1..3] |> List.iter (fun i -> i + 1 |> ignore)

for i in [1..3] do
    i + 1 |> ignore
```

19. Threading state through an iteration

The `fold` function is the most basic and powerful function in the collection arsenal. All other functions (other than generators like `unfold`) can be written in terms of it. See the examples below.

- `fold<'T,'State> : folder:('State -> 'T -> 'State) -> state:'State -> list:'T list -> 'State`. Applies a function to each element of the collection, threading an accumulator argument through the computation.
- `foldBack<'T,'State> : folder:('T -> 'State -> 'State) -> list:'T list -> state:'State -> 'State`. Applies a function to each element of the collection, starting from the end, threading an accumulator argument through the computation. WARNING: Watch out for using `Seq.foldBack` on infinite lists! The runtime will laugh at you ha ha ha and then go very quiet.

The `fold` function is often called "fold left" and `foldBack` is often called "fold right".

The `scan` function is like `fold` but returns the intermediate results and thus can be used to trace or monitor the iteration.

- `scan<'T,'State> : folder:('State -> 'T -> 'State) -> state:'State -> list:'T list -> 'State list`. Like `fold`, but returns both the intermediary and final results.
- `scanBack<'T,'State> : folder:('T -> 'State -> 'State) -> list:'T list -> state:'State -> 'State list`. Like `foldBack`, but returns both the intermediary and final results.

Just like the fold twins, `scan` is often called "scan left" and `scanBack` is often called "scan right".

Finally, `mapFold` combines `map` and `fold` into one awesome superpower. More complicated than using `map` and `fold` separately but also more efficient.

- `mapFold<'T,'State,'Result> : mapping:('State -> 'T -> 'Result * 'State) -> state:'State -> list:'T list -> 'Result list * 'State`. Combines map and fold. Builds a new collection whose elements are the results of applying the given function to each of the elements of the input collection. The function is also used to accumulate a final value.
- `mapFoldBack<'T,'State,'Result> : mapping:('T -> 'State -> 'Result * 'State) ->`

`list:'T list -> state:'State -> 'Result list * 'State`. Combines map and foldBack. Builds a new collection whose elements are the results of applying the given function to each of the elements of the input collection. The function is also used to accumulate a final value.

`fold` examples

One way of thinking about `fold` is that it is like `reduce` but with an extra parameter for the initial state:

```
["a";"b";"c"] |> List.fold (+) "hello: "
// "hello: abc"
// "hello: " + "a" + "b" + "c"

[1;2;3] |> List.fold (+) 10
// 16
// 10 + 1 + 2 + 3
```

As with `reduce`, `fold` and `foldBack` can give very different answers.

```
[1;2;3;4] |> List.fold (fun state x -> (state)*10 + x) 0
                                // state at each step
1                               // 1
(1)*10 + 2                      // 12
((1)*10 + 2)*10 + 3             // 123
(((1)*10 + 2)*10 + 3)*10 + 4    // 1234
// Final result is 1234
```

And here's the `foldBack` version:

```
List.foldBack (fun x state -> x + 10*(state)) [1;2;3;4] 0
                                // state at each step
4                               // 4
3 + 10*(4)                      // 43
2 + 10*(3 + 10*(4))             // 432
1 + 10*(2 + 10*(3 + 10*(4)))    // 4321
// Final result is 4321
```

Note that `foldBack` has a different parameter order to `fold`: the list is second last, and the initial state is last, which means that piping is not as convenient.

Recursing vs iterating

It's easy to get confused between `fold` vs. `foldBack`. I find it helpful to think of `fold` as being about *iteration* while `foldBack` is about *recursion*.

Let's say we want to calculate the sum of a list. The iterative way would be to use a for-loop. You start with a (mutable) accumulator and thread it through each iteration, updating it as you go.

```
let iterativeSum list =
    let mutable total = 0
    for e in list do
        total <- total + e
    total // return sum
```

On the other hand, the recursive approach says that if the list has a head and tail, calculate the sum of the tail (a smaller list) first, and then add the head to it.

Each time the tail gets smaller and smaller until it is empty, at which point you're done.

```
let rec recursiveSum list =
    match list with
    | [] ->
        0
    | head::tail ->
        head + (recursiveSum tail)
```

Which approach is better?

For aggregation, the iterative way is (`fold`) often easiest to understand. But for things like constructing new lists, the recursive way is (`foldBack`) is easier to understand.

For example, if we were going to going to create a function from scratch that turned each element into the corresponding string, we might write something like this:

```
let rec mapToString list =
    match list with
    | [] ->
        []
    | head::tail ->
        head.ToString() :: (mapToString tail)

[1..3] |> mapToString
// ["1"; "2"; "3"]
```

Using `foldBack` we can transfer that same logic "as is":

- action for empty list = `[]`

* action for non-empty list = `head.ToString() :: state`

Here is the resulting function:

```
let foldToString list =
    let folder head state =
        head.ToString() :: state
    List.foldBack folder list []

[1..3] |> foldToString
// ["1"; "2"; "3"]
```

On the other hand, a big advantage of `fold` is that it is easier to use "inline" because it plays better with piping.

Luckily, you can use `fold` (for list construction at least) just like `foldBack` as long as you reverse the list at the end.

```
// inline version of "foldToString"
[1..3]
|> List.fold (fun state head -> head.ToString() :: state) []
|> List.rev
// ["1"; "2"; "3"]
```

Using `fold` to implement other functions

As I mentioned above, `fold` is the core function for operating on lists and can emulate most other functions, although perhaps not as efficiently as a custom implementation.

For example, here is `map` implemented using `fold` :

```
/// map a function "f" over all elements
let myMap f list =
    // helper function
    let folder state head =
        f head :: state

    // main flow
    list
    |> List.fold folder []
    |> List.rev

[1..3] |> myMap (fun x -> x + 2)
// [3; 4; 5]
```

And here is `filter` implemented using `fold` :

```
/// return a new list of elements for which "pred" is true
let myFilter pred list =
    // helper function
    let folder state head =
        if pred head then
            head :: state
        else
            state

    // main flow
    list
    |> List.fold folder []
    |> List.rev

let isOdd n = (n%2=1)
[1..5] |> myFilter isOdd
// [1; 3; 5]
```

And of course, you can emulate the other functions in a similar way.

`scan` examples

Earlier, I showed an example of the intermediate steps of `fold`:

```
[1;2;3;4] |> List.fold (fun state x -> (state)*10 + x) 0
                                // state at each step
1                               // 1
(1)*10 + 2                      // 12
((1)*10 + 2)*10 + 3             // 123
(((1)*10 + 2)*10 + 3)*10 + 4    // 1234
// Final result is 1234
```

For that example, I had to manually calculate the intermediate states,

Well, if I had used `scan`, I would have got those intermediate states for free!

```
[1;2;3;4] |> List.scan (fun state x -> (state)*10 + x) 0
// accumulates from left ===> [0; 1; 12; 123; 1234]
```

`scanBack` works the same way, but backwards of course:

```
List.scanBack (fun x state -> (state)*10 + x) [1;2;3;4] 0
// [4321; 432; 43; 4; 0]  <=== accumulates from right
```

Just as with `foldBack` the parameter order for "scan right" is inverted compared with "scan left".

Truncating a string with `scan`

Here's an example where `scan` is useful. Say that you have a news site, and you need to make sure headlines fit into 50 chars.

You could just truncate the string at 50, but that would look ugly. Instead you want to have the truncation end at a word boundary.

Here's one way of doing it using `scan` :

- Split the headline into words.
- Use `scan` to concat the words back together, generating a list of fragments, each with an extra word added.
- Get the longest fragment under 50 chars.

```
// start by splitting the text into words
let text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod te
mpor."
let words = text.Split(' ')
// [|"Lorem"; "ipsum"; "dolor"; "sit"; ... ]

// accumulate a series of fragments
let fragments = words |> Seq.scan (fun frag word -> frag + " " + word) ""
(*
" Lorem"
" Lorem ipsum"
" Lorem ipsum dolor"
" Lorem ipsum dolor sit"
" Lorem ipsum dolor sit amet,"
etc
*)

// get the longest fragment under 50
let longestFragUnder50 =
    fragments
    |> Seq.takeWhile (fun s -> s.Length <= 50)
    |> Seq.last

// trim off the first blank
let longestFragUnder50Trimmed =
    longestFragUnder50 |> (fun s -> s.[1..])

// The result is:
//   "Lorem ipsum dolor sit amet, consectetur"
```

Note that I'm using `Seq.scan` rather than `Array.scan`. This does a lazy scan and avoids having to create fragments that are not needed.

Finally, here is the complete logic as a utility function:

```
// the whole thing as a function
let truncText max (text:string) =
    if text.Length <= max then
        text
    else
        text.Split(' ')
        |> Seq.scan (fun frag word -> frag + " " + word) ""
        |> Seq.takeWhile (fun s -> s.Length <= max-3)
        |> Seq.last
        |> (fun s -> s.[1..] + "...")

"a small headline" |> truncText 50
// "a small headline"

text |> truncText 50
// "Lorem ipsum dolor sit amet, consectetur..."
```

Yes, I know that there is a more efficient implementation than this, but I hope that this little example shows off the power of `scan`.

`mapFold` examples

The `mapFold` function can do a map and a fold in one step, which can be convenient on occasion.

Here's an example of combining an addition and a sum in one step using `mapFold`:

```
let add1 x = x + 1

// add1 using map
[1..5] |> List.map (add1)
// Result => [2; 3; 4; 5; 6]

// sum using fold
[1..5] |> List.fold (fun state x -> state + x) 0
// Result => 15

// map and sum using mapFold
[1..5] |> List.mapFold (fun state x -> add1 x, (state + x)) 0
// Result => ([2; 3; 4; 5; 6], 15)
```

20. Working with the index of each element

Often, you need the index of the element as you do an iteration. You could use a mutable counter, but why not sit back and let the library do the work for you?

- `mapi: mapping:(int -> 'T -> 'U) -> list:'T list -> 'U list`. Like `map`, but with the integer index passed to the function as well. See section 17 for more on `map`.
- `iteri: action:(int -> 'T -> unit) -> list:'T list -> unit`. Like `iter`, but with the integer index passed to the function as well. See section 18 for more on `iter`.
- `indexed: list:'T list -> (int * 'T) list`. Returns a new list whose elements are the corresponding elements of the input list paired with the index (from 0) of each element.

Usage examples

```
['a'..'c'] |> List.mapi (fun index ch -> sprintf "the %ith element is '%c'" index ch)
// ["the 0th element is 'a'"; "the 1th element is 'b'"; "the 2th element is 'c'"]

// with partial application
['a'..'c'] |> List.mapi (sprintf "the %ith element is '%c'")
// ["the 0th element is 'a'"; "the 1th element is 'b'"; "the 2th element is 'c'"]

['a'..'c'] |> List.iteri (printfn "the %ith element is '%c'")
(*
the 0th element is 'a'
the 1th element is 'b'
the 2th element is 'c'
*)
```

`indexed` generates a tuple with the index -- a shortcut for a specific use of `mapi`:

```
['a'..'c'] |> List.mapi (fun index ch -> (index, ch) )
// [(0, 'a'); (1, 'b'); (2, 'c')]

// "indexed" is a shorter version of above
['a'..'c'] |> List.indexed
// [(0, 'a'); (1, 'b'); (2, 'c')]
```

---

21. Transforming the whole collection to a different collection type

You often need to convert from one kind of collection to another. These functions do this.

The `ofXXX` functions are used to convert from `XXX` to the module type. For example, `List.ofArray` will turn an array into a list.

- (Except Array) `ofArray : array:'T[] -> 'T list`. Builds a new collection from the given array.
- (Except Seq) `ofSeq: source:seq<'T> -> 'T list`. Builds a new collection from the given enumerable object.
- (Except List) `ofList: source:'T list -> seq<'T>`. Builds a new collection from the given list.

The `toXXX` are used to convert from the module type to the type `XXX`. For example, `List.toArray` will turn an list into an array.

- (Except Array) `toArray: list:'T list -> 'T[]`. Builds an array from the given collection.
- (Except Seq) `toSeq: list:'T list -> seq<'T>`. Views the given collection as a sequence.
- (Except List) `toList: source:seq<'T> -> 'T list`. Builds a list from the given collection.

Usage examples

```
[1..5] |> List.toArray      // [|1; 2; 3; 4; 5|]
[1..5] |> Array.ofList      // [|1; 2; 3; 4; 5|]
// etc
```

Using sequences with disposables

One important use of these conversion functions is to convert a lazy enumeration (`seq`) to a fully evaluated collection such as `list`. This is particularly important when there is a disposable resource involved, such as file handle or database connection. If the sequence is not converted into a list you may encounter errors accessing the elements. See section 28 for more.

22. Changing the behavior of the collection as a whole

There are some special functions (for Seq only) that change the behavior of the collection as a whole.

- (Seq only) `cache: source:seq<'T> -> seq<'T>`. Returns a sequence that corresponds to

a cached version of the input sequence. This result sequence will have the same elements as the input sequence. The result can be enumerated multiple times. The input sequence will be enumerated at most once and only as far as is necessary.

- (Seq only) `readonly : source:seq<'T> -> seq<'T>`. Builds a new sequence object that delegates to the given sequence object. This ensures the original sequence cannot be rediscovered and mutated by a type cast.
- (Seq only) `delay : generator:(unit -> seq<'T>) -> seq<'T>`. Returns a sequence that is built from the given delayed specification of a sequence.

`cache` example

Here's an example of `cache` in use:

```
let uncachedSeq = seq {
    for i = 1 to 3 do
        printfn "Calculating %i" i
        yield i
    }

// iterate twice
uncachedSeq |> Seq.iter ignore
uncachedSeq |> Seq.iter ignore
```

The result of iterating over the sequence twice is as you would expect:

```
Calculating 1
Calculating 2
Calculating 3
Calculating 1
Calculating 2
Calculating 3
```

But if we cache the sequence...

```
let cachedSeq = uncachedSeq |> Seq.cache

// iterate twice
cachedSeq |> Seq.iter ignore
cachedSeq |> Seq.iter ignore
```

... then each item is only printed once:

```
Calculating 1
Calculating 2
Calculating 3
```

`readonly` example

Here's an example of `readonly` being used to hide the underlying type of the sequence:

```
// print the underlying type of the sequence
let printUnderlyingType (s:seq<_>) =
    let typeName = s.GetType().Name
    printfn "%s" typeName

[|1;2;3|] |> printUnderlyingType
// Int32[]

[|1;2;3|] |> Seq.readonly |> printUnderlyingType
// mkSeq@589   // a temporary type
```

`delay` example

Here's an example of `delay`.

```
let makeNumbers max =
    [ for i = 1 to max do
        printfn "Evaluating %d." i
        yield i ]

let eagerList =
    printfn "Started creating eagerList"
    let list = makeNumbers 5
    printfn "Finished creating eagerList"
    list

let delayedSeq =
    printfn "Started creating delayedSeq"
    let list = Seq.delay (fun () -> makeNumbers 5 |> Seq.ofList)
    printfn "Finished creating delayedSeq"
    list
```

If we run the code above, we find that just by creating `eagerList`, we print all the "Evaluating" messages. But creating `delayedSeq` does not trigger the list iteration.

```
Started creating eagerList
Evaluating 1.
Evaluating 2.
Evaluating 3.
Evaluating 4.
Evaluating 5.
Finished creating eagerList

Started creating delayedSeq
Finished creating delayedSeq
```

Only when the sequence is iterated over does the list creation happen:

```
eagerList |> Seq.take 3  // list already created
delayedSeq |> Seq.take 3 // list creation triggered
```

An alternative to using delay is just to embed the list in a `seq` like this:

```
let embeddedList = seq {
    printfn "Started creating embeddedList"
    yield! makeNumbers 5
    printfn "Finished creating embeddedList"
    }
```

As with `delayedSeq`, the `makeNumbers` function will not be called until the sequence is iterated over.

23. Working with two lists

If you have two lists, there are analogues of most of the common functions like map and fold.

- `map2: mapping:('T1 -> 'T2 -> 'U) -> list1:'T1 list -> list2:'T2 list -> 'U list`. Builds a new collection whose elements are the results of applying the given function to the corresponding elements of the two collections pairwise.
- `mapi2: mapping:(int -> 'T1 -> 'T2 -> 'U) -> list1:'T1 list -> list2:'T2 list -> 'U list`. Like `mapi`, but mapping corresponding elements from two lists of equal length.
- `iter2: action:('T1 -> 'T2 -> unit) -> list1:'T1 list -> list2:'T2 list -> unit`. Applies the given function to two collections simultaneously. The collections must have identical size.
- `iteri2: action:(int -> 'T1 -> 'T2 -> unit) -> list1:'T1 list -> list2:'T2 list ->`

`unit`. Like `iteri`, but mapping corresponding elements from two lists of equal length.

- `forall2: predicate:('T1 -> 'T2 -> bool) -> list1:'T1 list -> list2:'T2 list -> bool`. The predicate is applied to matching elements in the two collections up to the lesser of the two lengths of the collections. If any application returns false then the overall result is false, else true.
- `exists2: predicate:('T1 -> 'T2 -> bool) -> list1:'T1 list -> list2:'T2 list -> bool`. The predicate is applied to matching elements in the two collections up to the lesser of the two lengths of the collections. If any application returns true then the overall result is true, else false.
- `fold2<'T1,'T2,'State> : folder:('State -> 'T1 -> 'T2 -> 'State) -> state:'State -> list1:'T1 list -> list2:'T2 list -> 'State`. Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation.
- `foldBack2<'T1,'T2,'State> : folder:('T1 -> 'T2 -> 'State -> 'State) -> list1:'T1 list -> list2:'T2 list -> state:'State -> 'State`. Applies a function to corresponding elements of two collections, threading an accumulator argument through the computation.
- `compareWith: comparer:('T -> 'T -> int) -> list1:'T list -> list2:'T list -> int`. Compares two collections using the given comparison function, element by element. Returns the first non-zero result from the comparison function. If the end of a collection is reached it returns a -1 if the first collection is shorter and a 1 if the second collection is shorter.
- See also `append`, `concat`, and `zip` in section 26: combining and uncombining collections.

Usage examples

These functions are straightforward to use:

```
let intList1 = [2;3;4]
let intList2 = [5;6;7]

List.map2 (fun i1 i2 -> i1 + i2) intList1 intList2
//  [7; 9; 11]

// TIP use the ||> operator to pipe a tuple as two arguments
(intList1,intList2) ||> List.map2 (fun i1 i2 -> i1 + i2)
//  [7; 9; 11]

(intList1,intList2) ||> List.mapi2 (fun index i1 i2 -> index,i1 + i2)
 // [(0, 7); (1, 9); (2, 11)]

(intList1,intList2) ||> List.iter2 (printf "i1=%i i2=%i; ")
// i1=2 i2=5; i1=3 i2=6; i1=4 i2=7;

(intList1,intList2) ||> List.iteri2 (printf "index=%i i1=%i i2=%i; ")
// index=0 i1=2 i2=5; index=1 i1=3 i2=6; index=2 i1=4 i2=7;

(intList1,intList2) ||> List.forall2 (fun i1 i2 -> i1 < i2)
// true

(intList1,intList2) ||> List.exists2 (fun i1 i2 -> i1+10 > i2)
// true

(intList1,intList2) ||> List.fold2 (fun state i1 i2 -> (10*state) + i1 + i2) 0
// 801 = 234 + 567

List.foldBack2 (fun i1 i2 state -> i1 + i2 + (10*state)) intList1 intList2 0
// 1197 = 432 + 765

(intList1,intList2) ||> List.compareWith (fun i1 i2 -> i1.CompareTo(i2))
// -1

(intList1,intList2) ||> List.append
// [2; 3; 4; 5; 6; 7]

[intList1;intList2] |> List.concat
// [2; 3; 4; 5; 6; 7]

(intList1,intList2) ||> List.zip
// [(2, 5); (3, 6); (4, 7)]
```

Need a function that's not here?

By using `fold2` and `foldBack2` you can easily create your own functions. For example, some `filter2` functions can be defined like this:

```
/// Apply a function to each element in a pair
/// If either result passes, include that pair in the result
let filterOr2 filterPredicate list1 list2 =
    let pass e = filterPredicate e
    let folder e1 e2 state =
        if (pass e1) || (pass e2) then
            (e1,e2)::state
        else
            state
    List.foldBack2 folder list1 list2 ([])

/// Apply a function to each element in a pair
/// Only if both results pass, include that pair in the result
let filterAnd2 filterPredicate list1 list2 =
    let pass e = filterPredicate e
    let folder e1 e2 state =
        if (pass e1) && (pass e2) then
            (e1,e2)::state
        else
            state
    List.foldBack2 folder list1 list2 []

// test it
let startsWithA (s:string) = (s.[0] = 'A')
let strList1 = ["A1"; "A3"]
let strList2 = ["A2"; "B1"]

(strList1, strList2) ||> filterOr2 startsWithA
// [("A1", "A2"); ("A3", "B1")]
(strList1, strList2) ||> filterAnd2 startsWithA
// [("A1", "A2")]
```

See also section 25.

24. Working with three lists

If you have three lists, you only have one built-in function available. But see section 25 for an example of how you can build your own three-list functions.

- `map3: mapping:('T1 -> 'T2 -> 'T3 -> 'U) -> list1:'T1 list -> list2:'T2 list -> list3:'T3 list -> 'U list`. Builds a new collection whose elements are the results of applying the given function to the corresponding elements of the three collections simultaneously.
- See also `append`, `concat`, and `zip3` in section 26: combining and uncombining collections.

25. Working with more than three lists

If you are working with more than three lists, there are no built in functions for you.

If this happens infrequently, then you could just collapse the lists into a single tuple using `zip2` and/or `zip3` in succession, and then process that tuple using `map`.

Alternatively you can "lift" your function to the world of "zip lists" using applicatives.

```
let (<*>) fList xList =
    List.map2 (fun f x -> f x) fList xList

let (<!>) = List.map

let addFourParams x y z w =
    x + y + z + w

// lift "addFourParams" to List world and pass lists as parameters rather than ints
addFourParams <!> [1;2;3] <*> [1;2;3] <*> [1;2;3] <*> [1;2;3]
// Result = [4; 8; 12]
```

If that seems like magic, see this series for a explanation of what this code is doing.

26. Combining and uncombining collections

Finally, there are a number of functions that combine and uncombine collections.

- `append: list1:'T list -> list2:'T list -> 'T list`. Returns a new collection that contains the elements of the first collection followed by elements of the second.
- `@` is an infix version of `append` for lists.
- `concat: lists:seq<'T list> -> 'T list`. Builds a new collection whose elements are the results of applying the given function to the corresponding elements of the collections simultaneously.
- `zip: list1:'T1 list -> list2:'T2 list -> ('T1 * 'T2) list`. Combines two collections into a list of pairs. The two collections must have equal lengths.
- `zip3: list1:'T1 list -> list2:'T2 list -> list3:'T3 list -> ('T1 * 'T2 * 'T3) list`. Combines three collections into a list of triples. The collections must have equal lengths.
- (Except Seq) `unzip: list:('T1 * 'T2) list -> ('T1 list * 'T2 list)`. Splits a collection of pairs into two collections.
- (Except Seq) `unzip3: list:('T1 * 'T2 * 'T3) list -> ('T1 list * 'T2 list * 'T3`

`list)` . Splits a collection of triples into three collections.

Usage examples

These functions are straightforward to use:

```
List.append [1;2;3] [4;5;6]
// [1; 2; 3; 4; 5; 6]

[1;2;3] @ [4;5;6]
// [1; 2; 3; 4; 5; 6]

List.concat [ [1]; [2;3]; [4;5;6] ]
// [1; 2; 3; 4; 5; 6]

List.zip [1;2] [10;20]
// [(1, 10); (2, 20)]

List.zip3 [1;2] [10;20] [100;200]
// [(1, 10, 100); (2, 20, 200)]

List.unzip [(1, 10); (2, 20)]
// ([1; 2], [10; 20])

List.unzip3 [(1, 10, 100); (2, 20, 200)]
// ([1; 2], [10; 20], [100; 200])
```

Note that the `zip` functions require the lengths to be the same.

```
List.zip [1;2] [10]
// ArgumentException: The lists had different lengths.
```

---

27. Other array-only functions

Arrays are mutable, and therefore have some functions that are not applicable to lists and sequences.

- See the "sort in place" functions in section 15
- `Array.blit: source:'T[] -> sourceIndex:int -> target:'T[] -> targetIndex:int -> count:int -> unit` . Reads a range of elements from the first array and write them into the second.
- `Array.copy: array:'T[] -> 'T[]` . Builds a new array that contains the elements of the given array.

- `Array.fill: target:'T[] -> targetIndex:int -> count:int -> value:'T -> unit`. Fills a range of elements of the array with the given value.
- `Array.set: array:'T[] -> index:int -> value:'T -> unit`. Sets an element of an array.
- In addition to these, all the other BCL array functions are available as well.

I won't give examples. See the MSDN documentation.

28. Using sequences with disposables

One important use of conversion functions like `List.ofSeq` is to convert a lazy enumeration (`seq`) to a fully evaluated collection such as `list`. This is particularly important when there is a disposable resource involved such as file handle or database connection. If the sequence is not converted into a list while the resource is available you may encounter errors accessing the elements later, after the resource has been disposed.

This will be an extended example, so let's start with some helper functions that emulate a database and a UI:

```
// a disposable database connection
let DbConnection() =
    printfn "Opening connection"
    { new System.IDisposable with
        member this.Dispose() =
            printfn "Disposing connection" }

// read some records from the database
let readNCustomersFromDb dbConnection n =
    let makeCustomer i =
        sprintf "Customer %i" i

    seq {
        for i = 1 to n do
            let customer = makeCustomer i
            printfn "Loading %s from db" customer
            yield customer
        }

// show some records on the screen
let showCustomersinUI customers =
    customers |> Seq.iter (printfn "Showing %s in UI")
```

A naive implementation will cause the sequence to be evaluated *after* the connection is closed:

```
let readCustomersFromDb() =
    use dbConnection = DbConnection()
    let results = readNCustomersFromDb dbConnection 2
    results

let customers = readCustomersFromDb()
customers |> showCustomersinUI
```

The output is below. You can see that the connection is closed and only then is the sequence evaluated.

```
Opening connection
Disposing connection
Loading Customer 1 from db  // error! connection closed!
Showing Customer 1 in UI
Loading Customer 2 from db
Showing Customer 2 in UI
```

A better implementation will convert the sequence to a list while the connection is open, causing the sequence to be evaluated immediately:

```
let readCustomersFromDb() =
    use dbConnection = DbConnection()
    let results = readNCustomersFromDb dbConnection 2
    results |> List.ofSeq
    // Convert to list while connection is open

let customers = readCustomersFromDb()
customers |> showCustomersinUI
```

The result is much better. All the records are loaded before the connection is disposed:

```
Opening connection
Loading Customer 1 from db
Loading Customer 2 from db
Disposing connection
Showing Customer 1 in UI
Showing Customer 2 in UI
```

A third alternative is to embed the disposable in the sequence itself:

```
let readCustomersFromDb() =
    seq {
        // put disposable inside the sequence
        use dbConnection = DbConnection()
        yield! readNCustomersFromDb dbConnection 2
        }

let customers = readCustomersFromDb()
customers |> showCustomersinUI
```

The output shows that now the UI display is also done while the connection is open:

```
Opening connection
Loading Customer 1 from db
Showing Customer 1 in UI
Loading Customer 2 from db
Showing Customer 2 in UI
Disposing connection
```

This may be a bad thing (longer time for the connection to stay open) or a good thing (minimal memory use), depending on the context.

---

29. The end of the adventure

You made it to the end -- well done! Not really much of an adventure, though, was it? No dragons or anything. Nevertheless, I hope it was helpful.

As has been stressed many times before, F# is fundamentally a functional language at heart, yet the OO features have been nicely integrated and do not have a "tacked-on" feeling. As a result, it is quite viable to use F# just as an OO language, as an alternative to C#, say.

In this series, we'll look at how F# supports object-oriented classes and methods.

- Object-oriented programming in F#: Introduction. .
- Classes. .
- Inheritance and abstract classes. .
- Interfaces. .
- Object expressions. .

Object-oriented programming in F#: Introduction

In this series, we'll look at how F# supports object-oriented classes and methods.

Should you use object-oriented features at all?

As has been stressed many times before, F# is fundamentally a functional language at heart, yet the OO features have been nicely integrated and do not have a "tacked-on" feeling. As a result, it is quite viable to use F# just as an OO language, as an alternative to C#, say.

Whether to use the OO style or the functional style is, of course, up to you. Here are some arguments for and against.

Reasons in favor of using OO features:

- If you just want to do a direct port from C# without refactoring. (For more on this, there is a entire series on how to port from C# to F#.)
- If you want to use F# primarily as an OO language, as an alternative to C#.
- If you need to integrate with other .NET languages

Reasons against using OO features:

- If you are a beginner coming from an imperative language, classes can be a crutch that hinder your understanding of functional programming.
- Classes do not have the convenient "out of the box" features that the "pure" F# data types have, such as built-in equality and comparison, pretty printing, etc.
- Classes and methods do not play well with the type inference system and higher order functions (see discussion here), so using them heavily means that you are making it harder to benefit from the most powerful parts of F#.

In most cases, the best approach is a hybrid one, primarily using pure F# types and functions to benefit from type inference, but occasionally using interfaces and classes when polymorphism is needed.

Understanding the object-oriented features of F#

If you do decide to use the object-oriented features of F#, the following series of posts should cover everything you need to know to be productive with classes and methods in F#.

First up, how to create classes!

Classes

This post and the next will cover the basics of creating and using classes and methods in F#.

Defining a class

Just like all other data types in F#, class definitions start with the `type` keyword.

The thing that distinguishes them from other types is that classes always have some parameters passed in when they are created -- the constructor -- and so there are *always parentheses after the class name*.

Also, unlike other types, classes *must* have functions attached to them as members. This post will explain how you do this for classes, but for a general discussion of attaching functions to other types see the post on type extensions.

So, for example, if we want to have a class called `CustomerName` that requires three parameters to construct it, it would be written like this:

```
type CustomerName(firstName, middleInitial, lastName) =
    member this.FirstName = firstName
    member this.MiddleInitial = middleInitial
    member this.LastName = lastName
```

Let's compare this with the C# equivalent:

```
public class CustomerName
{
    public CustomerName(string firstName,
       string middleInitial, string lastName)
    {
        this.FirstName = firstName;
        this.MiddleInitial = middleInitial;
        this.LastName = lastName;
    }

    public string FirstName { get; private set; }
    public string MiddleInitial { get; private set; }
    public string LastName { get; private set; }
}
```

You can see that in the F# version, the primary constructor is embedded into the class declaration itself --- it is not a separate method. That is, the class declaration has the same parameters as the constructor, and the parameters automatically become immutable private fields that store the original values that were passed in.

So in the above example, because we declared the `CustomerName` class as:

```
type CustomerName(firstName, middleInitial, lastName)
```

therefore `firstName`, `middleInitial`, and `lastName` automatically became immutable private fields.

Specifying types in the constructor

You might not have noticed, but the `CustomerName` class defined above does not constrain the parameters to be strings, unlike the C# version. In general, type inference from usage will probably force the values to be strings, but if you do need to specify the types explicitly, you can do so in the usual way with a colon followed by the type name.

Here's a version of the class with explicit types in the constructor:

```
type CustomerName2(firstName:string,
                   middleInitial:string, lastName:string) =
    member this.FirstName = firstName
    member this.MiddleInitial = middleInitial
    member this.LastName = lastName
```

One little quirk about F# is that if you ever need to pass a tuple as a parameter to a constructor, you will have to annotate it explicitly, because the call to the constructor will look identical:

```
type NonTupledConstructor(x:int,y: int) =
    do printfn "x=%i y=%i" x y

type TupledConstructor(tuple:int * int) =
    let x,y = tuple
    do printfn "x=%i y=%i" x y

// calls look identical
let myNTC = new NonTupledConstructor(1,2)
let myTC = new TupledConstructor(1,2)
```

Class members

The example class above has three read-only instance properties. In F#, both properties and methods use the `member` keyword.

Also, in the example above, you see the word " `this` " in front of each member name. This is a "self-identifier" that can be used to refer to the current instance of the class. Every non-static member must have a self-identifier, even it is not used (as in the properties above). There is no requirement to use a particular word, just as long as it is consistent. You could use "this" or "self" or "me" or any other word that commonly indicates a self reference.

Understanding class signatures

When a class is compiled (or when you over hover the definition in the editor), you see the "class signature" for the class. For example, for the class definition:

```
type MyClass(intParam:int, strParam:string) =
    member this.Two = 2
    member this.Square x = x * x
```

the corresponding signature is:

```
type MyClass =
  class
    new : intParam:int * strParam:string -> MyClass
    member Square : x:int -> int
    member Two : int
  end
```

The class signature contains the signatures for all the constructors, methods and properties in the class. It is worth understanding what these signatures mean, because, just as with functions, you can understand what the class does by looking at them. It is also important because you will need to write these signatures when creating abstract methods and interfaces.

Method signatures

Method signatures such as are very similar to the signatures for standalone functions, except that the parameter names are part of the signature itself.

So in this case, the method signature is:

```
member Square : x:int -> int
```

And for comparison, the corresponding signature for a standalone function would be:

```
val Square : int -> int
```

Constructor signatures

Constructor signatures are always called `new`, but other than that, they look like a method signature.

Constructor signatures always take tuple values as their only parameter. In this case the tuple type is `int * string`, as you would expect. The return type is the class itself, again as you would expect.

Again, we can compare the constructor signature with a similar standalone function:

```
// class constructor signature
new : intParam:int * strParam:string -> MyClass

// standalone function signature
val new : int * string -> MyClass
```

Property signatures

Finally, property signatures such as `member Two : int` are very similar to the signatures for standalone simple values, except that no explicit value is given.

```
// member property
member Two : int

// standalone value
val Two : int = 2
```

Private fields and functions using "let" bindings

After the class declaration, you can optionally have a set of "let" bindings, typically used for defining private fields and functions.

Here's some sample code to demonstrate this:

```
type PrivateValueExample(seed) =

    // private immutable value
    let privateValue = seed + 1

    // private mutable value
    let mutable mutableValue = 42

    // private function definition
    let privateAddToSeed input =
        seed + input

    // public wrapper for private function
    member this.AddToSeed x =
        privateAddToSeed x

    // public wrapper for mutable value
    member this.SetMutableValue x =
        mutableValue <- x

// test
let instance = new PrivateValueExample(42)
printf "%i" (instance.AddToSeed 2)
instance.SetMutableValue 43
```

In the example above, there are three let bindings:

- `privateValue` is set to the initial seed plus 1
- `mutableValue` is set to 42
- The `privateAddToSeed` function uses the initial seed plus a parameter

Because they are let bindings, they are automatically private, so to access them externally, there must be a public member to act as a wrapper.

Note that the `seed` value passed into the constructor is also available as a private field, just like the let-bound values.

Mutable constructor parameters

Sometimes, you want a parameter passed to the constructor to be mutable. You cannot specify this in the parameter itself, so the standard technique is to create a mutable let-bound value and assign it from the parameter, as shown below:

```
type MutableConstructorParameter(seed) =
    let mutable mutableSeed = seed

    // public wrapper for mutable value
    member this.SetSeed x =
        mutableSeed <- x
```

In cases, like this, it is quite common to give the mutable value the same name as the parameter itself, like this:

```
type MutableConstructorParameter2(seed) =
    let mutable seed = seed // shadow the parameter

    // public wrapper for mutable value
    member this.SetSeed x =
        seed <- x
```

Additional constructor behavior with "do" blocks

In the `CustomerName` example earlier, the constructor just allowed some values to be passed in but didn't do anything else. However, in some cases, you might need to execute some code as part of the constructor. This is done using `do` blocks.

Here's an example:

```
type DoExample(seed) =
    let privateValue = seed + 1

    //extra code to be done at construction time
    do printfn "the privateValue is now %i" privateValue

// test
new DoExample(42)
```

The "do" code can also call any let-bound functions defined before it, as shown in this example:

```
type DoPrivateFunctionExample(seed) =
    let privateValue = seed + 1

    // some code to be done at construction time
    do printfn "hello world"

    // must come BEFORE the do block that calls it
    let printPrivateValue() =
        do printfn "the privateValue is now %i" privateValue

    // more code to be done at construction time
    do printPrivateValue()

// test
new DoPrivateFunctionExample(42)
```

Accessing the instance via "this" in a do block

One of the differences between the "do" and "let" bindings is that the "do" bindings can access the instance while "let" bindings cannot. This is because "let" bindings are actually evaluated before the constructor itself (similar to field initializers in C#), so the instance in a sense does not exist yet.

If you need to call members of the instance from a "do" block, you need some way to refer to the instance itself. This is again done using a "self-identifier", but this time it is attached to the class declaration itself.

```
type DoPublicFunctionExample(seed) as this =
    // Note the "this" keyword in the declaration

    let privateValue = seed + 1

    // extra code to be done at construction time
    do this.PrintPrivateValue()

    // member
    member this.PrintPrivateValue() =
        do printfn "the privateValue is now %i" privateValue

// test
new DoPublicFunctionExample(42)
```

In general though, it is not best practice to call members from constructors unless you have to (e.g. calling a virtual method). Better to call private let-bound functions, and if necessary, have the public members call those same private functions.

Methods

A method definition is very like a function definition, except that it has the `member` keyword and the self-identifier instead of just the `let` keyword.

Here are some examples:

```
type MethodExample() =

    // standalone method
    member this.AddOne x =
        x + 1

    // calls another method
    member this.AddTwo x =
        this.AddOne x |> this.AddOne

    // parameterless method
    member this.Pi() =
        3.14159

// test
let me = new MethodExample()
printfn "%i" <| me.AddOne 42
printfn "%i" <| me.AddTwo 42
printfn "%f" <| me.Pi()
```

You can see that, just like normal functions, methods can have parameters, call other methods, and be parameterless (or to be precise, take a unit parameter)

Tuple form vs. curried form

Unlike normal functions, methods with more than one parameter can be defined in two different ways:

- The curried form, where parameters are separated with spaces, and partial application is supported. (Why "curried"? See the explanation of currying.)
- The tuple form, where all the parameters as passed in at the same time, comma-separated, in a single tuple.

The curried approach is more functional, and the tuple approach is more object-oriented. Here is an example class with a method for each approach:

```
type TupleAndCurriedMethodExample() =

    // curried form
    member this.CurriedAdd x y =
        x + y

    // tuple form
    member this.TupleAdd(x,y) =
        x + y

// test
let tc = new TupleAndCurriedMethodExample()
printfn "%i" <| tc.CurriedAdd 1 2
printfn "%i" <| tc.TupleAdd(1,2)

// use partial application
let addOne = tc.CurriedAdd 1
printfn "%i" <| addOne 99
```

So which approach should you use?

The advantages of tuple form are:

- Compatible with other .NET code
- Supports named parameters and optional parameters
- Supports method overloads (multiple methods with the same name that differ only in their function signature)

On the other hand, the disadvantages of tuple form are:

- Doesn't support partial application
- Doesn't work well with higher order functions
- Doesn't work well with type inference

For a more detailed discussion on tuple form vs. curried form see the post on type extensions.

Let- bound functions in conjunction with class methods

A common pattern is to create let-bound functions that do all the heavy lifting, and then have the public methods call these internal functions directly. This has the benefit that the type inference works much better with functional-style code than with methods.

Here's an example:

```
type LetBoundFunctions() =

    let listReduce reducer list =
        list |> List.reduce reducer

    let reduceWithSum sum elem =
        sum + elem

    let sum list =
        list |> listReduce reduceWithSum

    // finally a public wrapper
    member this.Sum  = sum

// test
let lbf = new LetBoundFunctions()
printfn "Sum is %i" <| lbf.Sum [1..10]
```

For more details on how to do this, see this discussion.

Recursive methods

Unlike normal let-bound functions, methods that are recursive do not need the special `rec` keyword. Here's the boringly familiar Fibonacci function as a method:

```
type MethodExample() =

    // recursive method without "rec" keyword
    member this.Fib x =
        match x with
        | 0 | 1 -> 1
        | _ -> this.Fib (x-1) + this.Fib (x-2)

// test
let me = new MethodExample()
printfn "%i" <| me.Fib 10
```

Type annotation for methods

As usual, the types for a method's parameters and return value can normally be inferred by the compiler, but if you need to specify them, you do so in the same way that you would for a standard function:

```
type MethodExample() =
    // explicit type annotation
    member this.AddThree (x:int) :int =
        x + 3
```

Properties

Properties can be divided into three groups:

- Immutable properties, where there is a "get" but no "set".
- Mutable properties, where there is a "get" and also a (possibly private) "set".
- Write-only properties, where there is a "set" but no "get". These are so unusual that I won't discuss them here, but the MSDN documentation describes the syntax if you ever need it.

The syntax for immutable and mutable properties is slightly different.

For immutable properties, the syntax is simple. There is a "get" member that is similar to a standard "let" value binding. The expression on the right-hand side of the binding can be any standard expression, typically a combination of the constructor parameters, private let-bound fields, and private functions.

Here's an example:

```
type PropertyExample(seed) =
    // immutable property
    // using a constructor parameter
    member this.Seed = seed
```

For mutable properties however, the syntax is more complicated. You need to provide two functions, one to get and one to set. This is done by using the syntax:

```
with get() = ...
and set(value) = ...
```

Here's an example:

```
type PropertyExample(seed) =
    // private mutable value
    let mutable myProp = seed

    // mutable property
    // changing a private mutable value
    member this.MyProp
        with get() = myProp
        and set(value) =  myProp <- value
```

To make the set function private, use the keywords `private set` instead.

Automatic properties

Starting in VS2012, F# supports automatic properties, which remove the requirement to create a separate backing store for them.

To create an immutable auto property, use the syntax:

```
member val MyProp = initialValue
```

To create a mutable auto property, use the syntax:

```
member val MyProp = initialValue with get,set
```

Note that in this syntax there is a new keyword `val` and the self-identifier has gone.

Complete property example

Here's a complete example that demonstrates all the property types:

```
type PropertyExample(seed) =
    // private mutable value
    let mutable myProp = seed

    // private function
    let square x = x * x

    // immutable property
    // using a constructor parameter
    member this.Seed = seed

    // immutable property
    // using a private function
    member this.SeedSquared = square seed

    // mutable property
    // changing a private mutable value
    member this.MyProp
        with get() = myProp
        and set(value) =  myProp <- value

    // mutable property with private set
    member this.MyProp2
        with get() = myProp
        and private set(value) =  myProp <- value

    // automatic immutable property (in VS2012)
    member val ReadOnlyAuto = 1

    // automatic mutable property (in VS2012)
    member val ReadWriteAuto = 1 with get,set

// test
let pe = new PropertyExample(42)
printfn "%i" <| pe.Seed
printfn "%i" <| pe.SeedSquared
printfn "%i" <| pe.MyProp
printfn "%i" <| pe.MyProp2

// try calling set
pe.MyProp <- 43    // Ok
printfn "%i" <| pe.MyProp

// try calling private set
pe.MyProp2 <- 43   // Error
```

Properties vs. parameterless methods

At this point you might be confused by the difference between properties and parameterless methods. They look identical at first glance, but there is a subtle difference -- "parameterless" methods are not really parameterless; they always have a unit parameter.

Here's an example of the difference in both definition and usage:

```
type ParameterlessMethodExample() =
    member this.MyProp = 1    // No parens!
    member this.MyFunc() = 1  // Note the ()

// in use
let x = new ParameterlessMethodExample()
printfn "%i" <| x.MyProp      // No parens!
printfn "%i" <| x.MyFunc()    // Note the ()
```

You can also tell the difference by looking at the signature of the class definition

The class definition looks like this:

```
type ParameterlessMethodExample =
  class
    new : unit -> ParameterlessMethodExample
    member MyFunc : unit -> int
    member MyProp : int
  end
```

The method has signature `MyFunc : unit -> int` and the property has signature `MyProp : int`.

This is very similar to what the signatures would be if the function and property were declared standalone, outside of any class:

```
let MyFunc2() = 1
let MyProp2 = 1
```

The signatures for these would look like:

```
val MyFunc2 : unit -> int
val MyProp2 : int = 1
```

which is almost exactly the same.

If you are unclear on the difference and why the unit parameter is needed for the function, please read the discussion of parameterless methods.

Secondary constructors

In addition to the primary constructor embedded in its declaration, a class can have additional constructors. These are indicated by the `new` keyword and must call the primary constructor as their last expression.

```
type MultipleConstructors(param1, param2) =
    do printfn "Param1=%i Param12=%i" param1 param2

    // secondary constructor
    new(param1) =
        MultipleConstructors(param1,-1)

    // secondary constructor
    new() =
        printfn "Constructing..."
        MultipleConstructors(13,17)

// test
let mc1 = new MultipleConstructors(1,2)
let mc2 = new MultipleConstructors(42)
let mc3 = new MultipleConstructors()
```

Static members

Just as in C#, classes can have static members, and this is indicated with the `static` keyword. The `static` modifier comes before the member keyword.

Members which are static cannot have a self-identifier such as "this" because there is no instance for them to refer to.

```
type StaticExample() =
    member this.InstanceValue = 1
    static member StaticValue = 2  // no "this"

// test
let instance = new StaticExample()
printf "%i" instance.InstanceValue
printf "%i" StaticExample.StaticValue
```

Static constructors

There is no direct equivalent of a static constructor in F#, but you can create static let-bound values and static do-blocks that are executed when the class is first used.

```
type StaticConstructor() =

    // static field
    static let rand = new System.Random()

    // static do
    static do printfn "Class initialization!"

    // instance member accessing static field
    member this.GetRand() = rand.Next()
```

Accessibility of members

You can control the accessibility of a member with the standard .NET keywords `public`, `private` and `internal`. The accessibility modifiers come after the `member` keyword and before the member name.

Unlike C#, all class members are public by default, not private. This includes both properties and methods. However, non-members (e.g. let declarations) are private and cannot be made public.

Here's an example:

```
type AccessibilityExample() =
    member this.PublicValue = 1
    member private this.PrivateValue = 2
    member internal this.InternalValue = 3
// test
let a = new AccessibilityExample();
printf "%i" a.PublicValue
printf "%i" a.PrivateValue  // not accessible
```

For properties, if the set and get have different accessibilities, you can tag each part with a separate accessibility modifier.

```
type AccessibilityExample2() =
    let mutable privateValue = 42
    member this.PrivateSetProperty
        with get() =
            privateValue
        and private set(value) =
            privateValue <- value

// test
let a2 = new AccessibilityExample2();
printf "%i" a2.PrivateSetProperty  // ok to read
a2.PrivateSetProperty <- 43        // not ok to write
```

In practice, the "public get, private set" combination that is so common in C# is not generally needed in F#, because immutable properties can be defined more elegantly, as described earlier.

Tip: defining classes for use by other .NET code

If you are defining classes that need to interop with other .NET code, do not define them inside a module! Define them in a namespace instead, outside of any module.

The reason for this is that F# modules are exposed as static classes, and any F# classes defined inside a module are then defined as nested classes within the static class, which can mess up your interop. For example, some unit test runners don't like static classes.

F# classes which are defined outside a module are generated as normal top-level .NET classes, which is probably what you want. But remember that (as discussed in a previous post) if you don't declare a namespace specifically, your class will be placed in an automatically generated module, and will be nested without your knowledge.

Here's an example of two F# classes, one defined outside a module and one defined inside:

```
// Note: this code will not work in an .FSX script,
// only in an .FS source file.
namespace MyNamespace

type TopLevelClass() =
    let nothing = 0

module MyModule =

    type NestedClass() =
        let nothing = 0
```

And here's how the same code might look in C#:

```
namespace MyNamespace
{
  public class TopLevelClass
  {
  // code
  }

  public static class MyModule
  {
    public class NestedClass
    {
    // code
    }
  }
}
```

Constructing and using a class

Now that we have defined the class, how do we go about using it?

One way to create an instance of a class is straightfoward and just like C# -- use the `new` keyword and pass in the arguments to the constructor.

```
type MyClass(intParam:int, strParam:string) =
    member this.Two = 2
    member this.Square x = x * x

let myInstance = new MyClass(1,"hello")
```

However, in F#, the constructor is considered to be just another function, so you can normally eliminate the `new` and call the constructor function on its own, like this:

```
let myInstance2 = MyClass(1,"hello")
let point = System.Drawing.Point(1,2)   // works with .NET classes too!
```

In the case when you are creating a class that implements `IDisposable`, you will get a compiler warning if you do not use `new`.

```
let sr1 = System.IO.StringReader("")      // Warning
let sr2 = new System.IO.StringReader("")  // OK
```

This can be a useful reminder to use the `use` keyword instead of the `let` keyword for disposables. See the post on `use` for more.

Calling methods and properties

And once you have an instance, you can "dot into" the instance and use any methods and properties in the standard way.

```
myInstance.Two
myInstance.Square 2
```

We have seen many examples of member usage in the above discussion, and there's not too much to say about it.

Remember that, as discussed above, tuple-style methods and curried-style methods can be called in distinct ways:

```
type TupleAndCurriedMethodExample() =
    member this.TupleAdd(x,y) = x + y
    member this.CurriedAdd x y = x + y

let tc = TupleAndCurriedMethodExample()
tc.TupleAdd(1,2)      // called with parens
tc.CurriedAdd 1 2     // called without parens
2 |> tc.CurriedAdd 1  // partial application
```

Inheritance and abstract classes

This is a follow-on from the previous post on classes. This post will focus on inheritance in F#, and how to define and use abstract classes and interfaces.

Inheritance

To declare that a class inherits from another class, use the syntax:

```
type DerivedClass(param1, param2) =
   inherit BaseClass(param1)
```

The `inherit` keyword signals that `DerivedClass` inherits from `BaseClass`. In addition, some `BaseClass` constructor must be called at the same time.

It might be useful to compare F# with C# at this point. Here is some C# code for a very simple pair of classes.

```
public class MyBaseClass
{
    public MyBaseClass(int param1)
    {
        this.Param1 = param1;
    }
    public int Param1 { get; private set; }
}

public class MyDerivedClass: MyBaseClass
{
    public MyDerivedClass(int param1,int param2): base(param1)
    {
        this.Param2 = param2;
    }
    public int Param2 { get; private set; }
}
```

Note that the inheritance declaration `class MyDerivedClass: MyBaseClass` is distinct from the constructor which calls `base(param1)`.

Now here is the F# version:

```
type BaseClass(param1) =
   member this.Param1 = param1

type DerivedClass(param1, param2) =
   inherit BaseClass(param1)
   member this.Param2 = param2

// test
let derived = new DerivedClass(1,2)
printfn "param1=%O" derived.Param1
printfn "param2=%O" derived.Param2
```

Unlike C#, the inheritance part of the declaration, `inherit BaseClass(param1)` , contains both the class to inherit from *and* its constructor.

Abstract and virtual methods

Obviously, part of the point of inheritance is to be able to have abstract methods, virtual methods, and so on.

Defining abstract methods in the base class

In C#, an abstract method is indicated by the `abstract` keyword plus the method signature. In F#, it is the same concept, except that the way that function signatures are written in F# is quite different from C#.

```
// concrete function definition
let Add x y = x + y

// function signature
// val Add : int -> int -> int
```

So to define an abstract method, we use the signature syntax, along with the `abstract member` keywords:

```
type BaseClass() =
   abstract member Add: int -> int -> int
```

Notice that the equals sign has been replaced with a colon. This is what you would expect, as the equals sign is used for binding values, while the colon is used for type annotation.

Now, if you try to compile the code above, you will get an error! The compiler will complain that there is no implementation for the method. To fix this, you need to:

- provide a default implementation of the method, or
- tell the compiler that the class as whole is also abstract.

We'll look at both of these alternatives shortly.

Defining abstract properties

An abstract immutable property is defined in a similar way. The signature is just like that of a simple value.

```
type BaseClass() =
   abstract member Pi : float
```

If the abstract property is read/write, you add the get/set keywords.

```
type BaseClass() =
   abstract Area : float with get,set
```

Default implementations (but no virtual methods)

To provide a default implementation of an abstract method in the base class, use the `default` keyword instead of the `member` keyword:

```
// with default implementations
type BaseClass() =
   // abstract method
   abstract member Add: int -> int -> int
   // abstract property
   abstract member Pi : float

   // defaults
   default this.Add x y = x + y
   default this.Pi = 3.14
```

You can see that the default method is defined in the usual way, except for the use of `default` instead of `member`.

One major difference between F# and C# is that in C# you can combine the abstract definition and the default implementation into a single method, using the `virtual` keyword. In F#, you cannot. You must declare the abstract method and the default implementation separately. The `abstract member` has the signature, and the `default` has the implementation.

Abstract classes

If at least one abstract method does *not* have a default implementation, then the entire class is abstract, and you must indicate this by annotating it with the `AbstractClass` attribute.

```
[<AbstractClass>]
type AbstractBaseClass() =
   // abstract method
   abstract member Add: int -> int -> int

   // abstract immutable property
   abstract member Pi : float

   // abstract read/write property
   abstract member Area : float with get,set
```

If this is done, then the compiler will no longer complain about a missing implementation.

Overriding methods in subclasses

To override an abstract method or property in a subclass, use the `override` keyword instead of the `member` keyword. Other than that change, the overridden method is defined in the usual way.

```
[<AbstractClass>]
type Animal() =
   abstract member MakeNoise: unit -> unit

type Dog() =
   inherit Animal()
   override this.MakeNoise () = printfn "woof"

// test
// let animal = new Animal()  // error creating ABC
let dog = new Dog()
dog.MakeNoise()
```

And to call a base method, use the `base` keyword, just as in C#.

```
type Vehicle() =
   abstract member TopSpeed: unit -> int
   default this.TopSpeed() = 60

type Rocket() =
   inherit Vehicle()
   override this.TopSpeed() = base.TopSpeed() * 10

// test
let vehicle = new Vehicle()
printfn "vehicle.TopSpeed = %i" <| vehicle.TopSpeed()
let rocket = new Rocket()
printfn "rocket.TopSpeed = %i" <| rocket.TopSpeed()
```

Summary of abstract methods

Abstract methods are basically straightforward and similar to C#. There are only two areas that might be tricky if you are used to C#:

- You must understand how function signatures work and what their syntax is! For a detailed discussion see the post on function signatures.
- There is no all-in-one virtual method. You must define the abstract method and the default implementation separately.

Interfaces

Interfaces are available and fully supported in F#, but there are number of important ways in which their usage differs from what you might be used to in C#.

Defining interfaces

Defining an interface is similar to defining an abstract class. So similar, in fact, that you might easily get them confused.

Here's an interface definition:

```
type MyInterface =
   // abstract method
   abstract member Add: int -> int -> int

   // abstract immutable property
   abstract member Pi : float

   // abstract read/write property
   abstract member Area : float with get,set
```

And here's the definition for the equivalent abstract base class:

```
[<AbstractClass>]
type AbstractBaseClass() =
   // abstract method
   abstract member Add: int -> int -> int

   // abstract immutable property
   abstract member Pi : float

   // abstract read/write property
   abstract member Area : float with get,set
```

So what's the difference? As usual, all abstract members are defined by signatures only. The only difference seems to be the lack of the `[<AbstractClass>]` attribute.

But in the earlier discussion on abstract methods, we stressed that the `[<AbstractClass>]` attribute was required; the compiler would complain that the methods have no implementation otherwise. So how does the interface definition get away with it?

The answer is trivial, but subtle. *The interface has no constructor*. That is, it does not have any parentheses after the interface name:

```
type MyInterface =   // <- no parens!
```

That's it. Removing the parens will convert a class definition into an interface!

Explicit and implicit interface implementations

When it comes time to implement an interface in a class, F# is quite different from C#. In C#, you can add a list of interfaces to the class definition and implement the interfaces implicitly.

Not so in F#. In F#, all interfaces must be *explicitly* implemented.

In an explicit interface implementation, the interface members can only be accessed through an interface instance (e.g. by casting the class to the interface type). The interface members are not visible as part of the class itself.

C# has support for both explicit and implicit interface implementations, but almost always, the implicit approach is used, and many programmers are not even aware of explicit interfaces in C#.

Implementing interfaces in F#

So, how do you implement an interface in F#? You cannot just "inherit" from it, as you would an abstract base class. You have to provide an explicit implementation for each interface member using the syntax `interface XXX with`, as shown below:

```
type IAddingService =
    abstract member Add: int -> int -> int

type MyAddingService() =

    interface IAddingService with
        member this.Add x y =
            x + y

    interface System.IDisposable with
        member this.Dispose() =
            printfn "disposed"
```

The above code shows how the class `MyAddingService` explicitly implements the `IAddingService` and the `IDisposable` interfaces. After the required `interface XXX with` section, the members are implemented in the normal way.

(As an aside, note again that `MyAddingService()` has a constructor, while `IAddingService` does not.)

Using interfaces

So now let's try to use the adding service interface:

```
let mas = new MyAddingService()
mas.Add 1 2    // error
```

Immediately, we run into an error. It appears that the instance does not implement the `Add` method at all. Of course, what this really means is that we must cast it to the interface first using the `:>` operator:

```
// cast to the interface
let mas = new MyAddingService()
let adder = mas :> IAddingService
adder.Add 1 2  // ok
```

This might seem incredibly awkward, but in practice it is not a problem as in most cases the casting is done implicitly for you.

For example, you will typically be passing an instance to a function that specifies an interface parameter. In this case, the casting is done automatically:

```
// function that requires an interface
let testAddingService (adder:IAddingService) =
    printfn "1+2=%i" <| adder.Add 1 2  // ok

let mas = new MyAddingService()
testAddingService mas // cast automatically
```

And in the special case of `IDisposable`, the `use` keyword will also automatically cast the instance as needed:

```
let testDispose =
    use mas = new MyAddingService()
    printfn "testing"
    // Dispose() is called here
```

Object expressions

So as we saw in the previous post, implementing interfaces in F# is a bit more awkward than in C#. But F# has a trick up its sleeve, called "object expressions".

With object expressions, you can implement an interface on-the-fly, without having to create a class.

Implementing interfaces with object expressions

Object expressions are most commonly used to implement interfaces. To do this, you use the syntax `new MyInterface with ...`, and the wrap the whole thing in curly braces (one of the few uses for them in F#!)

Here is some example code that creates a number of objects, each of which implements `IDisposable`.

```
// create a new object that implements IDisposable
let makeResource name =
   { new System.IDisposable
     with member this.Dispose() = printfn "%s disposed" name }

let useAndDisposeResources =
    use r1 = makeResource "first resource"
    printfn "using first resource"
    for i in [1..3] do
        let resourceName = sprintf "\tinner resource %d" i
        use temp = makeResource resourceName
        printfn "\tdo something with %s" resourceName
    use r2 = makeResource "second resource"
    printfn "using second resource"
    printfn "done."
```

If you execute this code, you will see the output below. You can see that `Dispose()` is indeed being called when the objects go out of scope.

```
using first resource
    do something with   inner resource 1
    inner resource 1 disposed
    do something with   inner resource 2
    inner resource 2 disposed
    do something with   inner resource 3
    inner resource 3 disposed
using second resource
done.
second resource disposed
first resource disposed
```

We can take the same approach with the `IAddingService` and create one on the fly as well.

```
let makeAdder id =
   { new IAddingService with
     member this.Add x y =
         printfn "Adder%i is adding" id
         let result = x + y
         printfn "%i + %i = %i" x y result
         result
         }

let testAdders =
    for i in [1..3] do
        let adder = makeAdder i
        let result = adder.Add i i
        () //ignore result
```

Object expressions are extremely convenient, and can greatly reduce the number of classes you need to create if you are interacting with an interface heavy library.

In this series, you'll learn what computation expressions are, some common patterns, and how to make your own. In the process, we'll also look at continuations, the bind function, wrapper types, and more.

- Computation expressions: Introduction. Unwrapping the enigma....
- Understanding continuations. How 'let' works behind the scenes.
- Introducing 'bind'. Steps towards creating our own 'let!' .
- Computation expressions and wrapper types. Using types to assist the workflow.
- More on wrapper types. We discover that even lists can be wrapper types.
- Implementing a builder: Zero and Yield. Getting started with the basic builder methods.
- Implementing a builder: Combine. How to return multiple values at once.
- Implementing a builder: Delay and Run. Controlling when functions execute.
- Implementing a builder: Overloading. Stupid method tricks.
- Implementing a builder: Adding laziness. Delaying a workflow externally.
- Implementing a builder: The rest of the standard methods. Implementing While, Using, and exception handling.

Computation expressions: Introduction

By popular request, it is time to talk about the mysteries of computation expressions, what they are, and how they can be useful in practice (and I will try to avoid using the forbidden m-word).

In this series, you'll learn what computation expressions are, how to make your own, and what some common patterns involving them. In the process, we'll also look at continuations, the bind function, wrapper types, and more.

Background

Computation expressions seem to have a reputation for being abstruse and difficult to understand.

On one hand, they're easy enough to use. Anyone who has written much F# code has certainly used standard ones like `seq{...}` or `async{...}`.

But how do you make a new one of these things? How do they work behind the scenes?

Unfortunately, many explanations seem to make things even more confusing. There seems to be some sort of mental bridge that you have to cross. Once you are on the other side, it is all obvious, but to someone on this side, it is baffling.

If we turn for guidance to the official MSDN documention, it is explicit, but quite unhelpful to a beginner.

For example, it says that when you see the following code within a computation expression:

```
{| let! pattern = expr in cexpr |}
```

it is simply syntactic sugar for this method call:

```
builder.Bind(expr, (fun pattern -> {| cexpr |}))
```

But... what does this mean exactly?

I hope that by the end of this series, the documentation above will become obvious. Don't believe me? Read on!

Computation expressions in practice

Before going into the mechanics of computation expressions, let's look at a few trivial examples that show the same code before and after using computation expressions.

Let's start with a simple one. Let's say we have some code, and we want to log each step. So we define a little logging function, and call it after every value is created, like so:

```
let log p = printfn "expression is %A" p

let loggedWorkflow =
    let x = 42
    log x
    let y = 43
    log y
    let z = x + y
    log z
    //return
    z
```

If you run this, you will see the output:

```
expression is 42
expression is 43
expression is 85
```

Simple enough.

But it is annoying to have to explicitly write all the log statements each time. Is there a way to hide them?

Funny you should ask... A computation expression can do that. Here's one that does exactly the same thing.

First we define a new type called `LoggingBuilder` :

```
type LoggingBuilder() =
    let log p = printfn "expression is %A" p

    member this.Bind(x, f) =
        log x
        f x

    member this.Return(x) =
        x
```

Don't worry about what the mysterious `Bind` and `Return` are for yet -- they will be explained soon.

Next we create an instance of the type, `logger` in this case.

```
let logger = new LoggingBuilder()
```

So with this `logger` value, we can rewrite the original logging example like this:

```
let loggedWorkflow =
    logger
        {
        let! x = 42
        let! y = 43
        let! z = x + y
        return z
        }
```

If you run this, you get exactly the same output, but you can see that the use of the `logger{...}` workflow has allowed us to hide the repetitive code.

Safe division

Now let's look at an old chestnut.

Say that we want to divide a series of numbers, one after another, but one of them might be zero. How can we handle it? Throwing an exception is ugly. Sounds like a good match for the `option` type though.

First we need to create a helper function that does the division and gives us back an `int option`. If everything is OK, we get a `Some` and if the division fails, we get a `None`.

Then we can chain the divisions together, and after each division we need to test whether it failed or not, and keep going only if it was successful.

Here's the helper function first, and then the main workflow:

```
let divideBy bottom top =
    if bottom = 0
    then None
    else Some(top/bottom)
```

Note that I have put the divisor first in the parameter list. This is so we can write an expression like `12 |> divideBy 3`, which makes chaining easier.

Let's put it to use. Here is a workflow that attempts to divide a starting number three times:

```
let divideByWorkflow init x y z =
    let a = init |> divideBy x
    match a with
    | None -> None  // give up
    | Some a' ->    // keep going
        let b = a' |> divideBy y
        match b with
        | None -> None  // give up
        | Some b' ->    // keep going
            let c = b' |> divideBy z
            match c with
            | None -> None  // give up
            | Some c' ->    // keep going
                //return
                Some c'
```

And here it is in use:

```
let good = divideByWorkflow 12 3 2 1
let bad = divideByWorkflow 12 3 0 1
```

The `bad` workflow fails on the third step and returns `None` for the whole thing.

It is very important to note that the *entire workflow* has to return an `int option` as well. It can't just return an `int` because what would it evaluate to in the bad case? And can you see how the type that we used "inside" the workflow, the option type, has to be the same type that comes out finally at the end. Remember this point -- it will crop up again later.

Anyway, this continual testing and branching is really ugly! Does turning it into a computation expression help?

Once more we define a new type (`MaybeBuilder`) and make an instance of the type (`maybe`).

```
type MaybeBuilder() =

    member this.Bind(x, f) =
        match x with
        | None -> None
        | Some a -> f a

    member this.Return(x) =
        Some x

let maybe = new MaybeBuilder()
```

I have called this one `MaybeBuilder` rather than `divideByBuilder` because the issue of dealing with option types this way, using a computation expression, is quite common, and `maybe` is the standard name for this thing.

So now that we have defined the `maybe` workflow, let's rewrite the original code to use it.

```
let divideByWorkflow init x y z =
    maybe
        {
        let! a = init |> divideBy x
        let! b = a |> divideBy y
        let! c = b |> divideBy z
        return c
        }
```

Much, much nicer. The `maybe` expression has completely hidden the branching logic!

And if we test it we get the same result as before:

```
let good = divideByWorkflow 12 3 2 1
let bad = divideByWorkflow 12 3 0 1
```

Chains of "or else" tests

In the previous example of "divide by", we only wanted to continue if each step was successful.

But sometimes it is the other way around. Sometimes the flow of control depends on a series of "or else" tests. Try one thing, and if that succeeds, you're done. Otherwise try another thing, and if that fails, try a third thing, and so on.

Let's look at a simple example. Say that we have three dictionaries and we want to find the value corresponding to a key. Each lookup might succeed or fail, so we need to chain the lookups in a series.

```
let map1 = [ ("1","One"); ("2","Two") ] |> Map.ofList
let map2 = [ ("A","Alice"); ("B","Bob") ] |> Map.ofList
let map3 = [ ("CA","California"); ("NY","New York") ] |> Map.ofList

let multiLookup key =
    match map1.TryFind key with
    | Some result1 -> Some result1   // success
    | None ->   // failure
        match map2.TryFind key with
        | Some result2 -> Some result2 // success
        | None ->   // failure
            match map3.TryFind key with
            | Some result3 -> Some result3  // success
            | None -> None // failure
```

Because everything is an expression in F# we can't do an early return, we have to cascade all the tests in a single expression.

Here's how this might be used:

```
multiLookup "A" |> printfn "Result for A is %A"
multiLookup "CA" |> printfn "Result for CA is %A"
multiLookup "X" |> printfn "Result for X is %A"
```

It works fine, but can it be simplified?

Yes indeed. Here is an "or else" builder that allows us to simplify these kinds of lookups:

```
type OrElseBuilder() =
    member this.ReturnFrom(x) = x
    member this.Combine (a,b) =
        match a with
        | Some _ -> a  // a succeeds -- use it
        | None -> b    // a fails -- use b instead
    member this.Delay(f) = f()

let orElse = new OrElseBuilder()
```

Here's how the lookup code could be altered to use it:

```
let map1 = [ ("1","One"); ("2","Two") ] |> Map.ofList
let map2 = [ ("A","Alice"); ("B","Bob") ] |> Map.ofList
let map3 = [ ("CA","California"); ("NY","New York") ] |> Map.ofList

let multiLookup key = orElse {
    return! map1.TryFind key
    return! map2.TryFind key
    return! map3.TryFind key
    }
```

Again we can confirm that the code works as expected.

```
multiLookup "A" |> printfn "Result for A is %A"
multiLookup "CA" |> printfn "Result for CA is %A"
multiLookup "X" |> printfn "Result for X is %A"
```

Asynchronous calls with callbacks

Finally, let's look at callbacks. The standard approach for doing asynchronous operations in .NET is to use a AsyncCallback delegate which gets called when the async operation is complete.

Here is an example of how a web page might be downloaded using this technique:

```
open System.Net
let req1 = HttpWebRequest.Create("http://tryfsharp.org")
let req2 = HttpWebRequest.Create("http://google.com")
let req3 = HttpWebRequest.Create("http://bing.com")

req1.BeginGetResponse((fun r1 ->
    use resp1 = req1.EndGetResponse(r1)
    printfn "Downloaded %O" resp1.ResponseUri

    req2.BeginGetResponse((fun r2 ->
        use resp2 = req2.EndGetResponse(r2)
        printfn "Downloaded %O" resp2.ResponseUri

        req3.BeginGetResponse((fun r3 ->
            use resp3 = req3.EndGetResponse(r3)
            printfn "Downloaded %O" resp3.ResponseUri

            ),null) |> ignore
        ),null) |> ignore
    ),null) |> ignore
```

Lots of calls to `BeginGetResponse` and `EndGetResponse`, and the use of nested lambdas, makes this quite complicated to understand. The important code (in this case, just print statements) is obscured by the callback logic.

In fact, managing this cascading approach is always a problem in code that requires a chain of callbacks; it has even been called the "Pyramid of Doom" (although none of the solutions are very elegant, IMO).

Of course, we would never write that kind of code in F#, because F# has the `async` computation expression built in, which both simplifies the logic and flattens the code.

```
open System.Net
let req1 = HttpWebRequest.Create("http://tryfsharp.org")
let req2 = HttpWebRequest.Create("http://google.com")
let req3 = HttpWebRequest.Create("http://bing.com")

async {
    use! resp1 = req1.AsyncGetResponse()
    printfn "Downloaded %O" resp1.ResponseUri

    use! resp2 = req2.AsyncGetResponse()
    printfn "Downloaded %O" resp2.ResponseUri

    use! resp3 = req3.AsyncGetResponse()
    printfn "Downloaded %O" resp3.ResponseUri

    } |> Async.RunSynchronously
```

We'll see exactly how the `async` workflow is implemented later in this series.

Summary

So we've seen some very simple examples of computation expressions, both "before" and "after", and they are quite representative of the kinds of problems that computation expressions are useful for.

- In the logging example, we wanted to perform some side-effect between each step.
- In the safe division example, we wanted to handle errors elegantly so that we could focus on the happy path.
- In the multiple dictionary lookup example, we wanted to return early with the first success.
- And finally, in the async example, we wanted to hide the use of callbacks and avoid the "pyramid of doom".

What all the cases have in common is that the computation expression is "doing something behind the scenes" between each expression.

If you want a bad analogy, you can think of a computation expression as somewhat like a post-commit hook for SVN or git, or a database trigger that gets called on every update. And really, that's all that a computation expression is: something that allows you to sneak your own code in to be called *in the background*, which in turn allows you to focus on the important code in the foreground.

Why are they called "computation expressions"? Well, it's obviously some kind of expression, so that bit is obvious. I believe that the F# team did originally want to call it "expression-that-does-something-in-the-background-between-each-let" but for some reason, people thought that was a bit unwieldy, so they settled on the shorter name "computation expression" instead.

And as to the difference between a "computation expression" and a "workflow", I use *"computation expression"* to mean the `{...}` and `let!` syntax, and reserve *"workflow"* for particular implementations where appropriate. Not all computation expression implementations are workflows. For example, it is appropriate to talk about the "async workflow" or the "maybe workflow", but the "seq workflow" doesn't sound right.

In other words, in the following code, I would say that `maybe` is the workflow we are using, and the particular chunk of code `{ let! a = .... return c }` is the computation expression.

```
maybe
    {
    let! a = x |> divideBy y
    let! b = a |> divideBy w
    let! c = b |> divideBy z
    return c
    }
```

You probably want to start creating your own computation expressions now, but first we need to take a short detour into continuations. That's up next.

Update on 2015-01-11: I have removed the counting example that used a "state" computation expression. It was too confusing and distracted from the main concepts.

Understanding continuations

In the previous post we saw how some complex code could be condensed using computation expressions.

Here's the code before using a computation expression:

```
let log p = printfn "expression is %A" p

let loggedWorkflow =
    let x = 42
    log x
    let y = 43
    log y
    let z = x + y
    log z
    //return
    z
```

And here's the same code after using a computation expression:

```
let loggedWorkflow =
    logger
        {
        let! x = 42
        let! y = 43
        let! z = x + y
        return z
        }
```

The use of `let!` rather than a normal `let` is important. Can we emulate this ourselves so we can understand what is going on? Yes, but we need to understand continuations first.

Continuations

In imperative programming, we have the concept of "returning" from a function. When you call a function, you "go in", and then you "come out", just like pushing and popping a stack.

Here is some typical C# code which works like this. Notice the use of the `return` keyword.

```
public int Divide(int top, int bottom)
{
    if (bottom==0)
    {
        throw new InvalidOperationException("div by 0");
    }
    else
    {
        return top/bottom;
    }
}

public bool IsEven(int aNumber)
{
    var isEven = (aNumber % 2 == 0);
    return isEven;
}
```

You've seen this a million times, but there is a subtle point about this approach that you might not have considered: *the called function always decides what to do*.

For example, the implemention of `Divide` has decided that it is going to throw an exception. But what if I don't want an exception? Maybe I want a `nullable<int>`, or maybe I am going to display it on a screen as "#DIV/0". Why throw an exception that I am immediately going to have to catch? In other words, why not let the *caller* decide what should happen, rather the callee.

Similarly in the `IsEven` example, what am I going to do with the boolean return value? Branch on it? Or maybe print it in a report? I don't know, but again, rather than returning a boolean that the caller has to deal with, why not let the caller tell the callee what to do next?

So this is what continuations are. A **continuation** is simply a function that you pass into another function to tell it what to do next.

Here's the same C# code rewritten to allow the caller to pass in functions which the callee uses to handle each case. If it helps, you can think of this as somewhat analogous to a visitor pattern. Or maybe not.

```
public T Divide<T>(int top, int bottom, Func<T> ifZero, Func<int,T> ifSuccess)
{
    if (bottom==0)
    {
        return ifZero();
    }
    else
    {
        return ifSuccess( top/bottom );
    }
}

public T IsEven<T>(int aNumber, Func<int,T> ifOdd, Func<int,T> ifEven)
{
    if (aNumber % 2 == 0)
    {
        return ifEven(aNumber);
    }
    else
    {   return ifOdd(aNumber);
    }
}
```

Note that the C# functions have been changed to return a generic `T` now, and both continuations are a `Func` that returns a `T`.

Well, passing in lots of `Func` parameters always looks pretty ugly in C#, so it is not done very often. But passing functions is easy in F#, so let's see how this code ports over.

Here's the "before" code:

```
let divide top bottom =
    if (bottom=0)
    then invalidOp "div by 0"
    else (top/bottom)

let isEven aNumber =
    aNumber % 2 = 0
```

and here's the "after" code:

```
let divide ifZero ifSuccess top bottom =
    if (bottom=0)
    then ifZero()
    else ifSuccess (top/bottom)

let isEven ifOdd ifEven aNumber =
    if (aNumber % 2 = 0)
    then aNumber |> ifEven
    else aNumber |> ifOdd
```

A few things to note. First, you can see that I have put the extra functions (`ifZero` , etc) *first* in the parameter list, rather than last, as in the C# example. Why? Because I am probably going to want to use partial application.

And also, in the `isEven` example, I wrote `aNumber |> ifEven` and `aNumber |> ifOdd`. This makes it clear that we are piping the current value into the continuation and the continuation is always the very last step to be evaluated. *We will be using this exact same pattern later in this post, so make sure you understand what is going on here.*

Continuation examples

With the power of continuations at our disposal, we can use the same `divide` function in three completely different ways, depending on what the caller wants.

Here are three scenarios we can create quickly:

- pipe the result into a message and print it,
- convert the result to an option using `None` for the bad case and `Some` for the good case,
- or throw an exception in the bad case and just return the result in the good case.

```
// Scenario 1: pipe the result into a message
// ----------------------------------------
// setup the functions to print a message
let ifZero1 () = printfn "bad"
let ifSuccess1 x = printfn "good %i" x

// use partial application
let divide1  = divide ifZero1 ifSuccess1

//test
let good1 = divide1 6 3
let bad1 = divide1 6 0

// Scenario 2: convert the result to an option
// ----------------------------------------
// setup the functions to return an Option
let ifZero2() = None
let ifSuccess2 x = Some x
let divide2  = divide ifZero2 ifSuccess2

//test
let good2 = divide2 6 3
let bad2 = divide2 6 0

// Scenario 3: throw an exception in the bad case
// ----------------------------------------
// setup the functions to throw exception
let ifZero3() = failwith "div by 0"
let ifSuccess3 x = x
let divide3  = divide ifZero3 ifSuccess3

//test
let good3 = divide3 6 3
let bad3 = divide3 6 0
```

Notice that with this approach, the caller *never* has to catch an exception from `divide` anywhere. The caller decides whether an exception will be thrown, not the callee. So not only has the `divide` function become much more reusable in different contexts, but the cyclomatic complexity has just dropped a level as well.

The same three scenarios can be applied to the `isEven` implementation:

```
// Scenario 1: pipe the result into a message
// ----------------------------------------
// setup the functions to print a message
let ifOdd1 x = printfn "isOdd %i" x
let ifEven1 x = printfn "isEven %i" x

// use partial application
let isEven1  = isEven ifOdd1 ifEven1

//test
let good1 = isEven1 6
let bad1 = isEven1 5

// Scenario 2: convert the result to an option
// ----------------------------------------
// setup the functions to return an Option
let ifOdd2 _ = None
let ifEven2 x = Some x
let isEven2  = isEven ifOdd2 ifEven2

//test
let good2 = isEven2 6
let bad2 = isEven2 5

// Scenario 3: throw an exception in the bad case
// ----------------------------------------
// setup the functions to throw exception
let ifOdd3 _ = failwith "assert failed"
let ifEven3 x = x
let isEven3  = isEven ifOdd3 ifEven3

//test
let good3 = isEven3 6
let bad3 = isEven3 5
```

In this case, the benefits are subtler, but the same: the caller never had to handle booleans with an `if/then/else` anywhere. There is less complexity and less chance of error.

It might seem like a trivial difference, but by passing functions around like this, we can use all our favorite functional techniques such as composition, partial application, and so on.

We have also met continuations before, in the series on designing with types. We saw that their use enabled the caller to decide what would happen in case of possible validation errors in a constructor, rather than just throwing an exception.

```
type EmailAddress = EmailAddress of string

let CreateEmailAddressWithContinuations success failure (s:string) =
    if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then success (EmailAddress s)
        else failure "Email address must contain an @ sign"
```

The success function takes the email as a parameter and the error function takes a string. Both functions must return the same type, but the type is up to you.

And here is a simple example of the continuations in use. Both functions do a printf, and return nothing (i.e. unit).

```
// setup the functions
let success (EmailAddress s) = printfn "success creating email %s" s
let failure  msg = printfn "error creating email: %s" msg
let createEmail = CreateEmailAddressWithContinuations success failure

// test
let goodEmail = createEmail "x@example.com"
let badEmail = createEmail "example.com"
```

Continuation passing style

Using continuations like this leads to a style of programming called "continuation passing style" (or CPS), whereby *every* function is called with an extra "what to do next" function parameter.

To see the difference, let's look at the standard, direct style of programming.

When you use the direct style, you go "in" and "out" of functions, like this

```
call a function ->
   <- return from the function
call another function ->
   <- return from the function
call yet another function ->
   <- return from the function
```

In continuation passing style, on the other hand, you end up with a chain of functions, like this:

```
evaluate something and pass it into ->
   a function that evaluates something and passes it into ->
      another function that evaluates something and passes it into ->
         yet another function that evaluates something and passes it into ->
            ...etc...
```

There is obviously a big difference between the two styles.

In the direct style, there is a hierarchy of functions. The top level function is a sort of "master controller" who calls one subroutine, and then another, deciding when to branch, when to loop, and generally coordinating the control flow explicitly.

In the contination passing style, though, there is no "master controller". Instead there is a sort of "pipeline", not of data but of control flow, where the "function in charge" changes as the execution logic flows through the pipe.

If you have ever attached a event handler to a button click in a GUI, or used a callback with BeginInvoke, then you have used this style without being aware of it. And in fact, this style will be key to understanding the `async` workflow, which I'll discuss later in this series.

Continuations and 'let'

So how does all this fit in with `let` ?

Let's go back and revisit what 'let` actually does.

Remember that a (non-top-level) "let" can never be used in isolation -- it must always be part of a larger code block.

That is:

```
let x = someExpression
```

really means:

```
let x = someExpression in [an expression involving x]
```

And then every time you see the `x` in the second expression (the body expression), substitute it with the first expression (`someExpression`).

So for example, the expression:

```
let x = 42
let y = 43
let z = x + y
```

really means (using the verbose `in` keyword):

```
let x = 42 in
  let y = 43 in
    let z = x + y in
       z    // the result
```

Now funnily enough, a lambda looks very similar to a `let`:

```
fun x -> [an expression involving x]
```

and if we pipe in the value of `x` as well, we get the following:

```
someExpression |> (fun x -> [an expression involving x] )
```

Doesn't this look awfully like a `let` to you? Here is a let and a lambda side by side:

```
// let
let x = someExpression in [an expression involving x]

// pipe a value into a lambda
someExpression |> (fun x -> [an expression involving x] )
```

They both have an `x`, and a `someExpression`, and everywhere you see `x` in the body of the lambda you replace it with `someExpression`. Yes, the `x` and the `someExpression` are reversed in the lambda case, but otherwise it is basically the same thing as a `let`.

So, using this technique, we can rewrite the original example in this style:

```
42 |> (fun x ->
  43 |> (fun y ->
     x + y |> (fun z ->
       z)))
```

When it is written this way, you can see that we have transformed the `let` style into a continuation passing style!

- In the first line we have a value `42` -- what do we want to do with it? Let's pass it into a continuation, just as we did with the `isEven` function earlier. And in the context of the

continuation, we will relabel `42` as `x`.
- In the second line we have a value `43` -- what do we want to do with it? Let's pass it too into a continuation, calling it `y` in that context.
- In the third line we add the x and y together to create a new value. And what do we want to do with it? Another continuation, another label (`z`).
- Finally in the last line we are done and the whole expression evaluates to `z`.

Wrapping the continuation in a function

Let's get rid of the explicit pipe and write a little function to wrap this logic. We can't call it "let" because that is a reserved word, and more importantly, the parameters are backwards from 'let'. The "x" is on the right hand side, and the "someExpression" is on the left hand side. So we'll call it `pipeInto` for now.

The definition of `pipeInto` is really obvious:

```
let pipeInto (someExpression,lambda) =
    someExpression |> lambda
```

Note that we are passing both parameters in at once using a tuple rather than as two distinct parameters separated by whitespace. They will always come as a pair.

So, with this `pipeInto` function we can then rewrite the example once more as:

```
pipeInto (42, fun x ->
  pipeInto (43, fun y ->
    pipeInto (x + y, fun z ->
       z)))
```

or we can eliminate the indents and write it like this:

```
pipeInto (42, fun x ->
pipeInto (43, fun y ->
pipeInto (x + y, fun z ->
z)))
```

You might be thinking: so what? Why bother to wrap the pipe into a function?

The answer is that we can add *extra code* in the `pipeInto` function to do stuff "behine the scenes", just as in a computation expression.

The "logging" example revisited

Let's redefine `pipeInto` to add a little bit of logging, like this:

```
let pipeInto (someExpression,lambda) =
   printfn "expression is %A" someExpression
   someExpression |> lambda
```

Now... run that code again.

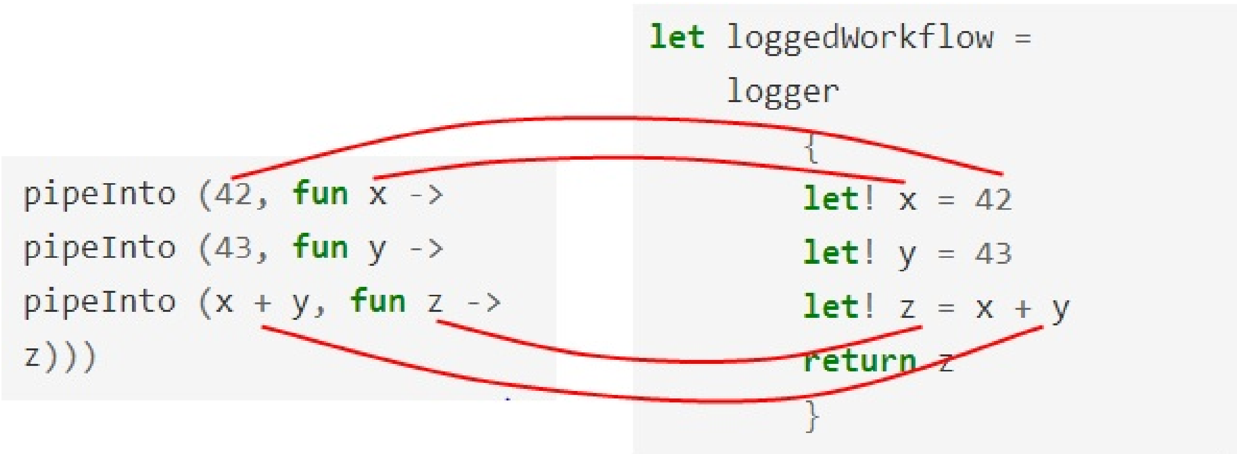
```
pipeInto (42, fun x ->
pipeInto (43, fun y ->
pipeInto (x + y, fun z ->
z
)))
```

What is the output?

```
expression is 42
expression is 43
expression is 85
```

This is exactly the same output as we had in the earlier implementations. We have created our own little computation expression workflow!

If we compare this side by side with the computation expression version, we can see that our homebrew version is very similar to the `let!`, except that we have the parameters reversed, and we have the explicit arrow for the continuation.



The "safe divide" example revisited

Let's do the same thing with the "safe divide" example. Here was the original code:

```
let divideBy bottom top =
    if bottom = 0
    then None
    else Some(top/bottom)

let divideByWorkflow x y w z =
    let a = x |> divideBy y
    match a with
    | None -> None  // give up
    | Some a' ->    // keep going
        let b = a' |> divideBy w
        match b with
        | None -> None  // give up
        | Some b' ->    // keep going
            let c = b' |> divideBy z
            match c with
            | None -> None  // give up
            | Some c' ->    // keep going
                //return
                Some c'
```

You should see now that this "stepped" style is an obvious clue that we really should be using continuations.

Let's see if we can add extra code to `pipeInto` to do the matching for us. The logic we want is:

- If the `someExpression` parameter is `None`, then don't call the continuation lambda.
- If the `someExpression` parameter is `Some`, then do call the continuation lambda, passing in the contents of the `Some`.

Here it is:

```
let pipeInto (someExpression,lambda) =
   match someExpression with
   | None ->
       None
   | Some x ->
       x |> lambda
```

With this new version of `pipeInto` we can rewrite the original code like this:

```
let divideByWorkflow x y w z =
    let a = x |> divideBy y
    pipeInto (a, fun a' ->
        let b = a' |> divideBy w
        pipeInto (b, fun b' ->
            let c = b' |> divideBy z
            pipeInto (c, fun c' ->
                Some c' //return
                )))
```

We can clean this up quite a bit.

First we can eliminate the `a`, `b` and `c`, and replace them with the `divideBy` expression directly. So that this:

```
let a = x |> divideBy y
pipeInto (a, fun a' ->
```

becomes just this:

```
pipeInto (x |> divideBy y, fun a' ->
```

Now we can relabel `a'` as just `a`, and so on, and we can also remove the stepped indentation, so that we get this:

```
let divideByResult x y w z =
    pipeInto (x |> divideBy y, fun a ->
    pipeInto (a |> divideBy w, fun b ->
    pipeInto (b |> divideBy z, fun c ->
    Some c //return
    )))
```

Finally, we'll create a little helper function called `return'` to wrap the result in an option. Putting it all together, the code looks like this:

```
let divideBy bottom top =
    if bottom = 0
    then None
    else Some(top/bottom)

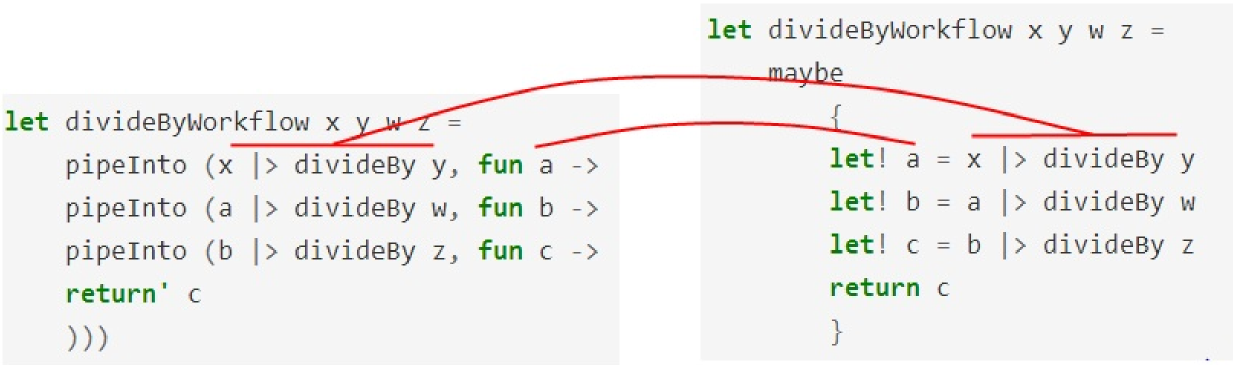
let pipeInto (someExpression,lambda) =
   match someExpression with
   | None ->
       None
   | Some x ->
       x |> lambda

let return' c = Some c

let divideByWorkflow x y w z =
    pipeInto (x |> divideBy y, fun a ->
    pipeInto (a |> divideBy w, fun b ->
    pipeInto (b |> divideBy z, fun c ->
    return' c
    )))

let good = divideByWorkflow 12 3 2 1
let bad = divideByWorkflow 12 3 0 1
```

Again, if we compare this side by side with the computation expression version, we can see that our homebrew version is identical in meaning. Only the syntax is different.



Summary

In this post, we talked about continuations and continuation passing style, and how we can think of `let` as a nice syntax for doing continuations behind scenes.

So now we have everything we need to start creating our *own* version of `let`. In the next post, we'll put this knowledge into practice.

Introducing 'bind'

In the last post we talked about how we can think of `let` as a nice syntax for doing continuations behind scenes. And we introduced a `pipeInto` function that allowed us to add hooks into the continuation pipeline.

Now we are ready to look at our first builder method, `Bind` , which formalizes this approach and is the core of any computation expression.

Introducing "Bind "

The MSDN page on computation expressions describes the `let!` expression as syntactic sugar for a `Bind` method. Let's look at this again:

Here's the `let!` expression documentation, along with a real example:

```
// documentation
{| let! pattern = expr in cexpr |}

// real example
let! x = 43 in some expression
```

And here's the `Bind` method documentation, along with a real example:

```
// documentation
builder.Bind(expr, (fun pattern -> {| cexpr |}))

// real example
builder.Bind(43, (fun x -> some expression))
```

Notice a few interesting things about this:

- `Bind` takes two parameters, an expression (`43`) and a lambda.
- The parameter of the lambda (`x`) is bound to the expression passed in as the first parameter. (In this case at least. More on this later.)
- The parameters of `Bind` are reversed from the order they are in `let!` .

So in other words, if we chain a number of `let!` expressions together like this:

```
let! x = 1
let! y = 2
let! z = x + y
```

the compiler converts it to calls to `Bind`, like this:

```
Bind(1, fun x ->
Bind(2, fun y ->
Bind(x + y, fun z ->
etc
```

I think you can see where we are going with this by now.

Indeed, our `pipeInto` function is exactly the same as the `Bind` method.

This is a key insight: *computation expressions are just a way to create nice syntax for something that we could do ourselves*.

A standalone bind function

Having a "bind" function like this is actually a standard functional pattern, and it is not dependent on computation expressions at all.

First, why is it called "bind"? Well, as we've seen, a "bind" function or method can be thought of as feeding an input value to a function. This is known as "binding" a value to the parameter of the function (recall that all functions have only one parameter).

So when you think of `bind` this this way, you can see that it is similar to piping or composition.

In fact, you can turn it into an infix operation like this:

```
let (>>=) m f = pipeInto(m,f)
```

By the way, this symbol ">>=" is the standard way of writing bind as an infix operator. If you ever see it used in other F# code, that is probably what it represents.

Going back to the safe divide example, we can now write the workflow on one line, like this:

```
let divideByWorkflow x y w z =
    x |> divideBy y >>= divideBy w >>= divideBy z
```

You might be wondering exactly how this is different from normal piping or composition? It's not immediately obvious.

The answer is twofold:

- First, the `bind` function has *extra* customized behavior for each situation. It is not a generic function, like pipe or composition.

- Second, the input type of the value parameter (`m` above) is not necessarily the same as the output type of the function parameter (`f` above), and so one of the things that bind does is handle this mismatch elegantly so that functions can be chained.

As we will see in the next post, bind generally works with some "wrapper" type. The value parameter might be of `WrapperType<TypeA>` , and then the signature of the function parameter of `bind` function is always `TypeA -> WrapperType<TypeB>` .

In the particular case of the `bind` for safe divide, the wrapper type is `Option` . The type of the value parameter (`m` above) is `Option<int>` and the signature of the function parameter (`f` above) is `int -> Option<int>` .

To see bind used in a different context, here is an example of the logging workflow expressed using a infix bind function:

```
let (>>=) m f =
    printfn "expression is %A" m
    f m

let loggingWorkflow =
    1 >>= (+) 2 >>= (*) 42 >>= id
```

In this case, there is no wrapper type. Everything is an `int` . But even so, `bind` has the special behavior that performs the logging behind the scenes.

Option.bind and the "maybe" workflow revisited

In the F# libraries, you will see `Bind` functions or methods in many places. Now you know what they are for!

A particularly useful one is `Option.bind` , which does exactly what we wrote by hand above, namely

- If the input parameter is `None` , then don't call the continuation function.
- If the input parameter is `Some` , then do call the continuation function, passing in the contents of the `Some` .

Here was our hand-crafted function:

```
let pipeInto (m,f) =
   match m with
   | None ->
       None
   | Some x ->
       x |> f
```

And here is the implementation of `Option.bind`:

```
module Option =
    let bind f m =
       match m with
       | None ->
           None
       | Some x ->
           x |> f
```

There is a moral in this -- don't be too hasty to write your own functions. There may well be library functions that you can reuse.

Here is the "maybe" workflow, rewritten to use `Option.bind`:

```
type MaybeBuilder() =
    member this.Bind(m, f) = Option.bind f m
    member this.Return(x) = Some x
```

Reviewing the different approaches so far

We've used four different approaches for the "safe divide" example so far. Let's put them together side by side and compare them once more.

Note: I have renamed the original `pipeInto` function to `bind`, and used `Option.bind` instead of our original custom implementation.

First the original version, using an explicit workflow:

```
module DivideByExplicit =

    let divideBy bottom top =
        if bottom = 0
        then None
        else Some(top/bottom)

    let divideByWorkflow x y w z =
        let a = x |> divideBy y
        match a with
        | None -> None  // give up
        | Some a' ->    // keep going
            let b = a' |> divideBy w
            match b with
            | None -> None  // give up
            | Some b' ->    // keep going
                let c = b' |> divideBy z
                match c with
                | None -> None  // give up
                | Some c' ->    // keep going
                    //return
                    Some c'
    // test
    let good = divideByWorkflow 12 3 2 1
    let bad = divideByWorkflow 12 3 0 1
```

Next, using our own version of "bind" (a.k.a. "pipeInto")

```
module DivideByWithBindFunction =

    let divideBy bottom top =
        if bottom = 0
        then None
        else Some(top/bottom)

    let bind (m,f) =
        Option.bind f m

    let return' x = Some x

    let divideByWorkflow x y w z =
        bind (x |> divideBy y, fun a ->
        bind (a |> divideBy w, fun b ->
        bind (b |> divideBy z, fun c ->
        return' c
        )))

    // test
    let good = divideByWorkflow 12 3 2 1
    let bad = divideByWorkflow 12 3 0 1
```

Next, using a computation expression:

```
module DivideByWithCompExpr =

    let divideBy bottom top =
        if bottom = 0
        then None
        else Some(top/bottom)

    type MaybeBuilder() =
        member this.Bind(m, f) = Option.bind f m
        member this.Return(x) = Some x

    let maybe = new MaybeBuilder()

    let divideByWorkflow x y w z =
        maybe
            {
            let! a = x |> divideBy y
            let! b = a |> divideBy w
            let! c = b |> divideBy z
            return c
            }

    // test
    let good = divideByWorkflow 12 3 2 1
    let bad = divideByWorkflow 12 3 0 1
```

And finally, using bind as an infix operation:

```
module DivideByWithBindOperator =

    let divideBy bottom top =
        if bottom = 0
        then None
        else Some(top/bottom)

    let (>>=) m f = Option.bind f m

    let divideByWorkflow x y w z =
        x |> divideBy y
        >>= divideBy w
        >>= divideBy z

    // test
    let good = divideByWorkflow 12 3 2 1
    let bad = divideByWorkflow 12 3 0 1
```

Bind functions turn out to be very powerful. In the next post we'll see that combining `bind` with wrapper types creates an elegant way of passing extra information around in the background.

Exercise: How well do you understand?

Before you move on to the next post, why don't you test yourself to see if you have understood everything so far?

Here is a little exercise for you.

Part 1 - create a workflow

First, create a function that parses a string into a int:

```
let strToInt str = ???
```

and then create your own computation expression builder class so that you can use it in a workflow, as shown below.

```
let stringAddWorkflow x y z =
    yourWorkflow
        {
        let! a = strToInt x
        let! b = strToInt y
        let! c = strToInt z
        return a + b + c
        }

// test
let good = stringAddWorkflow "12" "3" "2"
let bad = stringAddWorkflow "12" "xyz" "2"
```

Part 2 -- create a bind function

Once you have the first part working, extend the idea by adding two more functions:

```
let strAdd str i = ???
let (>>=) m f = ???
```

And then with these functions, you should be able to write code like this:

```
let good = strToInt "1" >>= strAdd "2" >>= strAdd "3"
let bad = strToInt "1" >>= strAdd "xyz" >>= strAdd "3"
```

Summary

Here's a summary of the points covered in this post:

- Computation expressions provide a nice syntax for continuation passing, hiding the chaining logic for us.
- `bind` is the key function that links the output of one step to the input of the next step.
- The symbol `>>=` is the standard way of writing bind as an infix operator.

Computation expressions and wrapper types

In the previous post, we were introduced to the "maybe" workflow, which allowed us to hide the messiness of chaining together option types.

A typical use of the "maybe" workflow looked something like this:

```
let result =
    maybe
        {
        let! anInt = expression of Option<int>
        let! anInt2 = expression of Option<int>
        return anInt + anInt2
        }
```

As we saw before, there is some apparently strange behavior going on here:

- In the `let!` lines, the expression on the *right* of the equals is an `int option`, but the value on the *left* is just an `int`. The `let!` has "unwrapped" the option before binding it to the value.
- And in the `return` line, the opposite occurs. The expression being returned is an `int`, but the value of the whole workflow (`result`) is an `int option`. That is, the `return` has "wrapped" the raw value back into an option.

We will follow up these observations in this post, and we will see that this leads to one of the major uses of computation expressions: namely, to implicitly unwrap and rewrap values that are stored in some sort of wrapper type.

Another example

Let's look at another example. Say that we are accessing a database, and we want to capture the result in a Success/Error union type, like this:

```
type DbResult<'a> =
    | Success of 'a
    | Error of string
```

We then use this type in our database access methods. Here are some very simple stubs to give you an idea of how the `DbResult` type might be used:

```
let getCustomerId name =
    if (name = "")
    then Error "getCustomerId failed"
    else Success "Cust42"

let getLastOrderForCustomer custId =
    if (custId = "")
    then Error "getLastOrderForCustomer failed"
    else Success "Order123"

let getLastProductForOrder orderId =
    if (orderId  = "")
    then Error "getLastProductForOrder failed"
    else Success "Product456"
```

Now let's say we want to chain these calls together. First get the customer id from the name, and then get the order for the customer id, and then get the product from the order.

Here's the most explicit way of doing it. As you can see, we have to have pattern matching at each step.

```
let product =
    let r1 = getCustomerId "Alice"
    match r1 with
    | Error _ -> r1
    | Success custId ->
        let r2 = getLastOrderForCustomer custId
        match r2 with
        | Error _ -> r2
        | Success orderId ->
            let r3 = getLastProductForOrder orderId
            match r3 with
            | Error _ -> r3
            | Success productId ->
                printfn "Product is %s" productId
                r3
```

Really ugly code. And the top-level flow has been submerged in the error handling logic.

Computation expressions to the rescue! We can write one that handles the branching of Success/Error behind the scenes:

```
type DbResultBuilder() =

    member this.Bind(m, f) =
        match m with
        | Error _ -> m
        | Success a ->
            printfn "\tSuccessful: %s" a
            f a

    member this.Return(x) =
        Success x

let dbresult = new DbResultBuilder()
```

And with this workflow, we can focus on the big picture and write much cleaner code:

```
let product' =
    dbresult {
        let! custId = getCustomerId "Alice"
        let! orderId = getLastOrderForCustomer custId
        let! productId = getLastProductForOrder orderId
        printfn "Product is %s" productId
        return productId
        }
printfn "%A" product'
```

And if there are errors, the workflow traps them nicely and tells us where the error was, as in this example below:

```
let product'' =
    dbresult {
        let! custId = getCustomerId "Alice"
        let! orderId = getLastOrderForCustomer "" // error!
        let! productId = getLastProductForOrder orderId
        printfn "Product is %s" productId
        return productId
        }
printfn "%A" product''
```

The role of wrapper types in workflows

So now we have seen two workflows (the `maybe` workflow and the `dbresult` workflow), each with their own corresponding wrapper type (`Option<T>` and `DbResult<T>` respectively).

These are not just special cases. In fact, *every* computation expression *must* have an associated wrapper type. And the wrapper type is often designed specifically to go hand-in-hand with the workflow that we want to manage.

The example above demonstrates this clearly. The `DbResult` type we created is more than just a simple type for return values; it actually has a critical role in the workflow by "storing" the current state of the workflow, and whether it is succeeding or failing at each step. By using the various cases of the type itself, the `dbresult` workflow can manage the transitions for us, hiding them from view and enabling us to focus on the big picture.

We'll learn how to design a good wrapper type later in the series, but first let's look at how they are manipulated.

Bind and Return and wrapper types

Let's look again at the definition of the `Bind` and `Return` methods of a computation expression.

We'll start off with the easy one, `Return`. The signature of `Return` as documented on MSDN is just this:

```
member Return : 'T -> M<'T>
```

In other words, for some type `T`, the `Return` method just wraps it in the wrapper type.

Note: In signatures, the wrapper type is normally called `M`, so `M<int>` is the wrapper type applied to `int` and `M<string>` is the wrapper type applied to `string`, and so on.

And we've seen two examples of this usage. The `maybe` workflow returns a `Some`, which is an option type, and the `dbresult` workflow returns `Success`, which is part of the `DbResult` type.

```
// return for the maybe workflow
member this.Return(x) =
    Some x

// return for the dbresult workflow
member this.Return(x) =
    Success x
```

Now let's look at `Bind`. The signature of `Bind` is this:

```
member Bind : M<'T> * ('T -> M<'U>) -> M<'U>
```

It looks complicated, so let's break it down. It takes a tuple `M<'T> * ('T -> M<'U>)` and returns a `M<'U>`, where `M<'U>` means the wrapper type applied to type `U`.

The tuple in turn has two parts:

- `M<'T>` is a wrapper around type `T`, and
- `'T -> M<'U>` is a function that takes a *unwrapped* `T` and creates a *wrapped* `U`.

In other words, what `Bind` does is:

- Take a *wrapped* value.
- Unwrap it and do any special "behind the scenes" logic.
- Then, optionally apply the function to the *unwrapped* value to create a new *wrapped* value.
- Even if the function is *not* applied, `Bind` must still return a *wrapped* `U`.

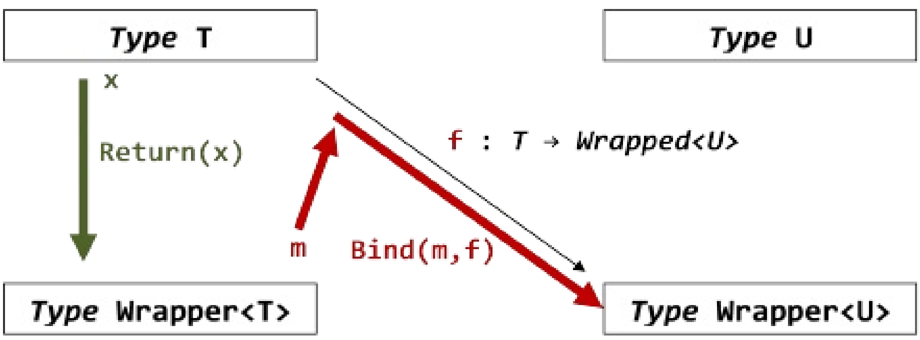
With this understanding, here are the `Bind` methods that we have seen already:

```
// return for the maybe workflow
member this.Bind(m,f) =
   match m with
   | None -> None
   | Some x -> f x

// return for the dbresult workflow
member this.Bind(m, f) =
    match m with
    | Error _ -> m
    | Success x ->
        printfn "\tSuccessful: %s" x
        f x
```

Look over this code and make sure that you understand why these methods do indeed follow the pattern described above.

Finally, a picture is always useful. Here is a diagram of the various types and functions:



- For `Bind`, we start with a wrapped value (`m` here), unwrap it to a raw value of type

`T`, and then (maybe) apply the function `f` to it to get a wrapped value of type `U`.
- For `Return`, we start with a value (`x` here), and simply wrap it.

The type wrapper is generic

Note that all the functions use generic types (`T` and `U`) other than the wrapper type itself, which must be the same throughout. For example, there is nothing stopping the `maybe` binding function from taking an `int` and returning a `Option<string>`, or taking a `string` and then returning an `Option<bool>`. The only requirement is that it always return an `Option<something>`.

To see this, we can revisit the example above, but rather than using strings everywhere, we will create special types for the customer id, order id, and product id. This means that each step in the chain will be using a different type.

We'll start with the types again, this time defining `CustomerId`, etc.

```
type DbResult<'a> =
    | Success of 'a
    | Error of string

type CustomerId =  CustomerId of string
type OrderId =  OrderId of int
type ProductId =  ProductId of string
```

The code is almost identical, except for the use of the new types in the `Success` line.

```
let getCustomerId name =
    if (name = "")
    then Error "getCustomerId failed"
    else Success (CustomerId "Cust42")

let getLastOrderForCustomer (CustomerId custId) =
    if (custId = "")
    then Error "getLastOrderForCustomer failed"
    else Success (OrderId 123)

let getLastProductForOrder (OrderId orderId) =
    if (orderId  = 0)
    then Error "getLastProductForOrder failed"
    else Success (ProductId "Product456")
```

Here's the long-winded version again.

```
let product =
    let r1 = getCustomerId "Alice"
    match r1 with
    | Error e -> Error e
    | Success custId ->
        let r2 = getLastOrderForCustomer custId
        match r2 with
        | Error e -> Error e
        | Success orderId ->
            let r3 = getLastProductForOrder orderId
            match r3 with
            | Error e -> Error e
            | Success productId ->
                printfn "Product is %A" productId
                r3
```

There are a couple of changes worth discussing:

- First, the `printfn` at the bottom uses the "%A" format specifier rather than "%s". This is required because the `ProductId` type is a union type now.
- More subtly, there seems to be unnecessary code in the error lines. Why write `| Error e -> Error e`? The reason is that the incoming error that is being matched against is of type `DbResult<CustomerId>` or `DbResult<OrderId>`, but the *return* value must be of type `DbResult<ProductId>`. So, even though the two `Error`s look the same, they are actually of different types.

Next up, the builder, which hasn't changed at all except for the `| Error e -> Error e` line.

```
type DbResultBuilder() =

    member this.Bind(m, f) =
        match m with
        | Error e -> Error e
        | Success a ->
            printfn "\tSuccessful: %A" a
            f a

    member this.Return(x) =
        Success x

let dbresult = new DbResultBuilder()
```

Finally, we can use the workflow as before.

```
let product' =
    dbresult {
        let! custId = getCustomerId "Alice"
        let! orderId = getLastOrderForCustomer custId
        let! productId = getLastProductForOrder orderId
        printfn "Product is %A" productId
        return productId
        }
printfn "%A" product'
```

At each line, the returned value is of a *different* type (`DbResult<CustomerId>`, `DbResult<OrderId>`, etc), but because they have the same wrapper type in common, the bind works as expected.

And finally, here's the workflow with an error case.

```
let product'' =
    dbresult {
        let! custId = getCustomerId "Alice"
        let! orderId = getLastOrderForCustomer (CustomerId "") //error
        let! productId = getLastProductForOrder orderId
        printfn "Product is %A" productId
        return productId
        }
printfn "%A" product''
```

Composition of computation expressions

We've seen that every computation expression *must* have an associated wrapper type. This wrapper type is used in both `Bind` and `Return`, which leads to a key benefit:

- *the output of a `Return` can be fed to the input of a `Bind`*

In other words, because a workflow returns a wrapper type, and because `let!` consumes a wrapper type, you can put a "child" workflow on the right hand side of a `let!` expression.

For example, say that you have a workflow called `myworkflow`. Then you can write the following:

```
let subworkflow1 = myworkflow { return 42 }
let subworkflow2 = myworkflow { return 43 }

let aWrappedValue =
    myworkflow {
        let! unwrappedValue1 = subworkflow1
        let! unwrappedValue2 = subworkflow2
        return unwrappedValue1 + unwrappedValue2
        }
```

Or you can even "inline" them, like this:

```
let aWrappedValue =
    myworkflow {
        let! unwrappedValue1 = myworkflow {
            let! x = myworkflow { return 1 }
            return x
            }
        let! unwrappedValue2 = myworkflow {
            let! y = myworkflow { return 2 }
            return y
            }
        return unwrappedValue1 + unwrappedValue2
        }
```

If you have used the `async` workflow, you probably have done this already, because an async workflow typically contains other asyncs embedded in it:

```
let a =
    async {
        let! x = doAsyncThing  // nested workflow
        let! y = doNextAsyncThing x // nested workflow
        return x + y
    }
```

Introducing "ReturnFrom"

We have been using `return` as a way of easily wrapping up an unwrapped return value.

But sometimes we have a function that already returns a wrapped value, and we want to return it directly. `return` is no good for this, because it requires an unwrapped type as input.

The solution is a variant on `return` called `return!`, which takes a *wrapped type* as input and returns it.

The corresponding method in the "builder" class is called `ReturnFrom`. Typically the implementation just returns the wrapped type "as is" (although of course, you can always add extra logic behind the scenes).

Here is a variant on the "maybe" workflow to show how it can be used:

```
type MaybeBuilder() =
    member this.Bind(m, f) = Option.bind f m
    member this.Return(x) =
        printfn "Wrapping a raw value into an option"
        Some x
    member this.ReturnFrom(m) =
        printfn "Returning an option directly"
        m

let maybe = new MaybeBuilder()
```

And here it is in use, compared with a normal `return`.

```
// return an int
maybe { return 1  }

// return an Option
maybe { return! (Some 2)  }
```

For a more realistic example, here is `return!` used in conjunction with `divideBy`:

```
// using return
maybe
    {
    let! x = 12 |> divideBy 3
    let! y = x |> divideBy 2
    return y  // return an int
    }

// using return!
maybe
    {
    let! x = 12 |> divideBy 3
    return! x |> divideBy 2  // return an Option
    }
```

Summary

This post introduced wrapper types and how they related to `Bind`, `Return` and `ReturnFrom`, the core methods of any builder class.

In the next post, we'll continue to look at wrapper types, including using lists as wrapper types.

More on wrapper types

In the previous post, we looked at the concept of "wrapper types" and their relation to computation expressions. In this post, we'll investigate what types are suitable for being wrapper types.

What kinds of types can be wrapper types?

If every computation expression must have an associated wrapper type, then what kinds of type can be used as wrapper types? Are there any special constraints or limitations that apply?

There is one general rule, which is:

- **Any type with a generic parameter can be used as a wrapper type**

So for example, you can use `Option<T>` , `DbResult<T>` , etc., as wrapper types, as we have seen. And you can use wrapper types that restrict the type parameter, such as `Vector<int>` .

But what about other generic types like `List<T>` or `IEnumerable<T>` ? Surely they can't be used? Actually, yes, they *can* be used! We'll see how shortly.

Can non-generic wrapper types work?

Is it possible to use a wrapper type that does *not* have a generic parameter?

For example, we saw in an earlier example an attempt to do addition on strings, like this: `"1" + "2"` . Can't we be clever and treat `string` as a wrapper type for `int` in this case? That would be cool, yes?

Let's try. We can use the signatures of `Bind` and `Return` to guide our implementation.

- `Bind` takes a tuple. The first part of the tuple is the wrapped type (`string` in this case), and the second part of the tuple is a function that takes an unwrapped type and converts it to a wrapped type. In this case, that would be `int -> string` .
- `Return` takes an unwrapped type (`int` in this case) and converts it to a wrapped type. So in this case, the signature of `Return` would be `int -> string` .

How does this guide the implementation?

- The implementation of the "rewrapping" function, `int -> string`, is easy. It is just "toString" on an int.
- The bind function has to unwrap a string to an int, and then pass it to the function. We can use `int.Parse` for that.
- But what happens if the bind function *can't* unwrap a string, because it is not a valid number? In this case, the bind function *must* still return a wrapped type (a string), so we can just return a string such as "error".

Here's the implementation of the builder class:

```
type StringIntBuilder() =

    member this.Bind(m, f) =
        let b,i = System.Int32.TryParse(m)
        match b,i with
        | false,_ -> "error"
        | true,i -> f i

    member this.Return(x) =
        sprintf "%i" x

let stringint = new StringIntBuilder()
```

Now we can try using it:

```
let good =
    stringint {
        let! i = "42"
        let! j = "43"
        return i+j
        }
printfn "good=%s" good
```

And what happens if one of the strings is invalid?

```
let bad =
    stringint {
        let! i = "42"
        let! j = "xxx"
        return i+j
        }
printfn "bad=%s" bad
```

That looks really good -- we can treat strings as ints inside our workflow!

But hold on, there is a problem.

Let's say we give the workflow an input, unwrap it (with `let!`) and then immediately rewrap it (with `return`) without doing anything else. What should happen?

```
let g1 = "99"
let g2 = stringint {
            let! i = g1
            return i
            }
printfn "g1=%s g2=%s" g1 g2
```

No problem. The input `g1` and the output `g2` are the same value, as we would expect.

But what about the error case?

```
let b1 = "xxx"
let b2 = stringint {
            let! i = b1
            return i
            }
printfn "b1=%s b2=%s" b1 b2
```

In this case we have got some unexpected behavior. The input `b1` and the output `b2` are *not* the same value. We have introduced an inconsistency.

Would this be a problem in practice? I don't know. But I would avoid it and use a different approach, like options, that are consistent in all cases.

Rules for workflows that use wrapper types

Here's a question? What is the difference between these two code fragments, and should they behave differently?

```
// fragment before refactoring
myworkflow {
    let wrapped = // some wrapped value
    let! unwrapped = wrapped
    return unwrapped
    }

// refactored fragment
myworkflow {
    let wrapped = // some wrapped value
    return! wrapped
    }
```

The answer is no, they should not behave differently. The only difference is that in the second example, the `unwrapped` value has been refactored away and the `wrapped` value is returned directly.

But as we just saw in the previous section, you can get inconsistencies if you are not careful. So, any implementation you create should be sure to follow some standard rules, which are:

Rule 1: If you start with an unwrapped value, and then you wrap it (using `return`), then unwrap it (using `bind`), you should always get back the original unwrapped value.

This rule and the next are about not losing information as you wrap and unwrap the values. Obviously, a sensible thing to ask, and required for refactoring to work as expected.

In code, this would be expressed as something like this:

```
myworkflow {
    let originalUnwrapped = something

    // wrap it
    let wrapped = myworkflow { return originalUnwrapped }

    // unwrap it
    let! newUnwrapped = wrapped

    // assert they are the same
    assertEqual newUnwrapped originalUnwrapped
    }
```

Rule 2: If you start with a wrapped value, and then you unwrap it (using `bind`), then wrap it (using `return`), you should always get back the original wrapped value.

This is the rule that the `stringInt` workflow broke above. As with rule 1, this should obviously be a requirement.

In code, this would be expressed as something like this:

```
myworkflow {
    let originalWrapped = something

    let newWrapped = myworkflow {

        // unwrap it
        let! unwrapped = originalWrapped

        // wrap it
        return unwrapped
        }

    // assert they are the same
    assertEqual newWrapped originalWrapped
    }
```

Rule 3: If you create a child workflow, it must produce the same result as if you had "inlined" the logic in the main workflow.

This rule is required for composition to behave properly, and again, "extraction" refactoring will only work correctly if this is true.

In general, you will get this for free if you follow some guidelines (which will be explained in a later post).

In code, this would be expressed as something like this:

```
// inlined
let result1 = myworkflow {
    let! x = originalWrapped
    let! y = f x  // some function on x
    return! g y   // some function on y
    }

// using a child workflow ("extraction" refactoring)
let result2 = myworkflow {
    let! y = myworkflow {
        let! x = originalWrapped
        return! f x // some function on x
        }
    return! g y     // some function on y
    }

// rule
assertEqual result1 result2
```

Lists as wrapper types

I said earlier that types like `List<T>` or `IEnumerable<T>` can be used as wrapper types. But how can this be? There is no one-to-one correspondence between the wrapper type and the unwrapped type!

This is where the "wrapper type" analogy becomes a bit misleading. Instead, let's go back to thinking of `bind` as a way of connecting the output of one expression with the input of another.

As we have seen, the `bind` function "unwraps" the type, and applies the continuation function to the unwrapped value. But there is nothing in the definition that says that there has to be only *one* unwrapped value. There is no reason that we can't apply the continuation function to each item of the list in turn.

In other words, we should be able to write a `bind` that takes a list and a continuation function, where the continuation function processes one element at a time, like this:

```
bind( [1;2;3], fun elem -> // expression using a single element )
```

And with this concept, we should be able to chain some binds together like this:

```
let add =
    bind( [1;2;3], fun elem1 ->
    bind( [10;11;12], fun elem2 ->
        elem1 + elem2
    ))
```

But we've missed something important. The continuation function passed into `bind` is required to have a certain signature. It takes an unwrapped type, but it produces a *wrapped* type.

In other words, the continuation function must *always create a new list* as its result.

```
bind( [1;2;3], fun elem -> // expression using a single element, returning a list )
```

And the chained example would have to be written like this, with the `elem1 + elem2` result turned into a list:

```
let add =
    bind( [1;2;3], fun elem1 ->
    bind( [10;11;12], fun elem2 ->
        [elem1 + elem2] // a list!
    ))
```

So the logic for our bind method now looks like this:

```
let bind(list,f) =
    // 1) for each element in list, apply f
    // 2) f will return a list (as required by its signature)
    // 3) the result is a list of lists
```

We have another issue now. `Bind` itself must produce a wrapped type, which means that the "list of lists" is no good. We need to turn them back into a simple "one-level" list.

But that is easy enough -- there is a list module function that does just that, called `concat`.

So putting it together, we have this:

```
let bind(list,f) =
    list
    |> List.map f
    |> List.concat

let added =
    bind( [1;2;3], fun elem1 ->
    bind( [10;11;12], fun elem2 ->
//       elem1 + elem2    // error.
        [elem1 + elem2]   // correctly returns a list.
    ))
```

Now that we understand how the `bind` works on its own, we can create a "list workflow".

- `Bind` applies the continuation function to each element of the passed in list, and then flattens the resulting list of lists into a one-level list. `List.collect` is a library function that does exactly that.
- `Return` converts from unwrapped to wrapped. In this case, that just means wrapping a single element in a list.

```
type ListWorkflowBuilder() =

    member this.Bind(list, f) =
        list |> List.collect f

    member this.Return(x) =
        [x]

let listWorkflow = new ListWorkflowBuilder()
```

Here is the workflow in use:

```
let added =
    listWorkflow {
        let! i = [1;2;3]
        let! j = [10;11;12]
        return i+j
        }
printfn "added=%A" added

let multiplied =
    listWorkflow {
        let! i = [1;2;3]
        let! j = [10;11;12]
        return i*j
        }
printfn "multiplied=%A" multiplied
```

And the results show that every element in the first collection has been combined with every element in the second collection:

```
val added : int list = [11; 12; 13; 12; 13; 14; 13; 14; 15]
val multiplied : int list = [10; 11; 12; 20; 22; 24; 30; 33; 36]
```

That's quite amazing really. We have completely hidden the list enumeration logic, leaving just the workflow itself.

Syntactic sugar for "for"

If we treat lists and sequences as a special case, we can add some nice syntactic sugar to replace `let!` with something a bit more natural.

What we can do is replace the `let!` with a `for..in..do` expression:

```
// let version
let! i = [1;2;3] in [some expression]

// for..in..do version
for i in [1;2;3] do [some expression]
```

Both variants mean exactly the same thing, they just look different.

To enable the F# compiler to do this, we need to add a `For` method to our builder class. It generally has exactly the same implementation as the normal `Bind` method, but is required to accept a sequence type.

```
type ListWorkflowBuilder() =

    member this.Bind(list, f) =
        list |> List.collect f

    member this.Return(x) =
        [x]

    member this.For(list, f) =
        this.Bind(list, f)

let listWorkflow = new ListWorkflowBuilder()
```

And here is how it is used:

```
let multiplied =
    listWorkflow {
        for i in [1;2;3] do
        for j in [10;11;12] do
        return i*j
        }
printfn "multiplied=%A" multiplied
```

LINQ and the "list workflow"

Does the `for element in collection do` look familiar? It is very close to the `from element in collection ...` syntax used by LINQ. And indeed LINQ uses basically the same technique to convert from a query expression syntax like `from element in collection ...` to actual method calls behine the scenes.

In F#, as we saw, the `bind` uses the `List.collect` function. The equivalent of `List.collect` in LINQ is the `SelectMany` extension method. And once you understand how `SelectMany` works, you can implement the same kinds of queries yourself. Jon Skeet has written a helpful blog post explaining this.

The identity "wrapper type"

So we've seen a number of wrapper types in this post, and have said that *every* computation expression *must* have an associated wrapper type.

But what about the logging example in the previous post? There was no wrapper type there. There was a `let!` that did things behind the scenes, but the input type was the same as the output type. The type was left unchanged.

The short answer to this is that you can treat any type as its own "wrapper". But there is another, deeper way to understand this.

Let's step back and consider what a wrapper type definition like `List<T>` really means.

If you have a type such as `List<T>`, it is in fact not a "real" type at all. `List<int>` is a real type, and `List<string>` is a real type. But `List<T>` on its own is incomplete. It is missing the parameter it needs to become a real type.

One way to think about `List<T>` is that it is a *function*, not a type. It is a function in the abstract world of types, rather than the concrete world of normal values, but just like any function it maps values to other values, except in this case, the input values are types (say `int` or `string`) and the output values are other types (`List<int>` and `List<string>`). And like any function it takes a parameter, in this case a "type parameter". Which is why the concept that .NET developers call "generics" is known as "parametric polymorphism" in computer science terminology.

Once we grasp the concept of functions that generate one type from another type (called "type constructors"), we can see that what we really mean by a "wrapper type" is just a type constructor.

But if a "wrapper type" is just a function that maps one type to another type, surely a function that maps a type to the *same* type fits into this category? And indeed it does. The "identity" function for types fits our definition and can be used as a wrapper type for computation expressions.

Going back to some real code then, we can define the "identity workflow" as the simplest possible implementation of a workflow builder.

```
type IdentityBuilder() =
    member this.Bind(m, f) = f m
    member this.Return(x) = x
    member this.ReturnFrom(x) = x

let identity = new IdentityBuilder()

let result = identity {
    let! x = 1
    let! y = 2
    return x + y
    }
```

With this in place, you can see that the logging example discussed earlier is just the identity workflow with some logging added in.

Summary

Another long post, and we covered a lot of topics, but I hope that the role of wrapper types is now clearer. We will see how the wrapper types can be used in practice when we come to look at common workflows such as the "writer workflow" and the "state workflow" later in this series.

Here's a summary of the points covered in this post:

- A major use of computation expressions is to unwrap and rewrap values that are stored in some sort of wrapper type.
- You can easily compose computation expressions, because the output of a `Return` can be fed to the input of a `Bind`.
- Every computation expression *must* have an associated wrapper type.
- Any type with a generic parameter can be used as a wrapper type, even lists.
- When creating workflows, you should ensure that your implementation conforms to the three sensible rules about wrapping and unwrapping and composition.

Implementing a builder: Zero and Yield

Having covered bind and continuations, and the use of wrapper types, we're finally ready to take on the full set of methods associated with "builder" classes.

If you look at the MSDN documentation, you'll see not just `Bind` and `Return`, but also other strangely named methods like `Delay` and `Zero`. What are *they* for? That's what this and the next few posts will answer.

The plan of action

To demonstrate how to create a builder class, we will create a custom workflow which uses all of the possible builder methods.

But rather than starting at the top and trying to explain what these methods mean without context, we'll work from the bottom up, starting with a simple workflow and adding methods only as needed to solve a problem or an error. In the process, you'll come to understand how F# processes computation expressions in detail.

The outline of this process is:

- Part 1: In this first part, we'll look at what methods are needed for a basic workflow. We'll introduce `Zero`, `Yield`, `Combine` and `For`.
- Part 2: Next, we'll look at how to delay the execution of your code, so that it is only evaluated when needed. We'll introduce `Delay` and `Run`, and look at lazy computations.
- Part 3: Finally, we'll cover the rest of the methods: `While`, `Using`, and exception handling.

Before we get started

Before we dive into creating the workflow, here are some general comments.

The documentation for computation expressions

First, as you might have noticed, the MSDN documentation for computation expressions is meagre at best, and although not inaccurate, can be misleading. For example, the signatures of the builder methods are *more* flexible than they appear to be, and this can be

used to implement some features that might not be obvious if you work from the documentation alone. We will show an example of this later.

If you want more detailed documentation, there are two sources I can recommend. For an detailed overview of the concepts behind computation expressions, a great resource is the paper "The F# Expression Zoo" by Tomas Petricek and Don Syme. And for the most accurate up-to-date technical documentation, you should read the F# language specification, which has a section on computation expressions.

Wrapped and unwrapped types

When you are trying to understand the signatures as documented, remember that what I have been calling the "unwrapped" type is normally written as `'T` and the "wrapped" type is normally written `M<'T>`. That is, when you see that the `Return` method has the signature `'T -> M<'T>` it means `Return` takes an unwrapped type and returns a wrapped type.

As I have in the earlier posts in this series, I will continue to use "unwrapped" and "wrapped" to describe the relationship between these types, but as we move forward these terms will be stretched to the breaking point, so I will also start using other terminology, such as "computation type" instead of "wrapped type". I hope that when we reach this point, the reason for the change will be clear and understandable.

Also, in my examples, I will generally try to keep things simple by using code such as:

```
let! x = ...wrapped type value...
```

But this is actually an oversimplification. To be precise, the "x" can be any *pattern* not just a single value, and the "wrapped type" value can, of course, be an *expression* that evaluates to a wrapped type. The MSDN documentation uses this more precise approach. It uses "pattern" and "expression" in the definitions, such as `let! pattern = expr in cexpr`.

Here are some examples of using patterns and expressions in a `maybe` computation expression, where `Option` is the wrapped type, and the right hand side expressions are `options`:

```
// let! pattern = expr in cexpr
maybe {
    let! x,y = Some(1,2)
    let! head::tail = Some( [1;2;3] )
    // etc
    }
```

Having said this, I will continue to use the oversimplified examples, so as not to add extra complication to an already complicated topic!

Implementing special methods in the builder class (or not)

The MSDN documentation shows that each special operation (such as `for..in`, or `yield`) is translated into one or more calls to methods in the builder class.

There is not always a one-to-one correspondence, but generally, to support the syntax for a special operation, you *must* implement a corresponding method in the builder class, otherwise the compiler will complain and give you an error.

On the other hand, you do *not* need to implement every single method if you don't need the syntax. For example, we have already implemented the `maybe` workflow quite nicely by only implementing the two methods `Bind` and `Return`. We don't need to implement `Delay`, `Use`, and so on, if we don't need to use them.

To see what happens if you have not implemented a method, let's try to use the `for..in..do` syntax in our `maybe` workflow like this:

```
maybe { for i in [1;2;3] do i }
```

We will get the compiler error:

```
This control construct may only be used if the computation expression builder defines
a 'For' method
```

Sometimes you get will errors that might be cryptic unless you know what is going on behind the scenes. For example, if you forget to put `return` in your workflow, like this:

```
maybe { 1 }
```

You will get the compiler error:

```
This control construct may only be used if the computation expression builder defines
a 'Zero' method
```

You might be asking: what is the `Zero` method? And why do I need it? The answer to that is coming right up.

Operations with and without '!'

Obviously, many of the special operations come in pairs, with and without a "!" symbol. For example: `let` and `let!` (pronounced "let-bang"), `return` and `return!`, `yield` and `yield!` and so on.

The difference is easy to remember when you realize that the operations *without* a "!" always have *unwrapped* types on the right hand side, while the ones *with* a "!" always have *wrapped* types.

So for example, using the `maybe` workflow, where `Option` is the wrapped type, we can compare the different syntaxes:

```
let x = 1             // 1 is an "unwrapped" type
let! x = (Some 1)     // Some 1 is a "wrapped" type
return 1              // 1 is an "unwrapped" type
return! (Some 1)      // Some 1 is a "wrapped" type
yield 1               // 1 is an "unwrapped" type
yield! (Some 1)       // Some 1 is a "wrapped" type
```

The "!" versions are particularly important for composition, because the wrapped type can be the result of *another* computation expression of the same type.

```
let! x = maybe {...)       // "maybe" returns a "wrapped" type

// bind another workflow of the same type using let!
let! aMaybe = maybe {...)  // create a "wrapped" type
return! aMaybe             // return it

// bind two child asyncs inside a parent async using let!
let processUri uri = async {
    let! html = webClient.AsyncDownloadString(uri)
    let! links = extractLinks html
    ... etc ...
    }
```

Diving in - creating a minimal implementation of a workflow

Let's start! We'll begin by creating a minimal version of the "maybe" workflow (which we'll rename as "trace") with every method instrumented, so we can see what is going on. We'll use this as our testbed throughout this post.

Here's the code for the first version of the `trace` workflow:

```
type TraceBuilder() =
    member this.Bind(m, f) =
        match m with
        | None ->
            printfn "Binding with None. Exiting."
        | Some a ->
            printfn "Binding with Some(%A). Continuing" a
        Option.bind f m

    member this.Return(x) =
        printfn "Returning a unwrapped %A as an option" x
        Some x

    member this.ReturnFrom(m) =
        printfn "Returning an option (%A) directly" m
        m

// make an instance of the workflow
let trace = new TraceBuilder()
```

Nothing new here, I hope. We have already seen all these methods before.

Now let's run some sample code through it:

```
trace {
    return 1
    } |> printfn "Result 1: %A"

trace {
    return! Some 2
    } |> printfn "Result 2: %A"

trace {
    let! x = Some 1
    let! y = Some 2
    return x + y
    } |> printfn "Result 3: %A"

trace {
    let! x = None
    let! y = Some 1
    return x + y
    } |> printfn "Result 4: %A"
```

Everything should work as expected, in particular, you should be able to see that the use of `None` in the 4th example caused the next two lines (`let! y = ... return x+y`) to be skipped and the result of the whole expression was `None` .

Introducing "do!"

Our expression supports `let!`, but what about `do!`?

In normal F#, `do` is just like `let`, except that the expression doesn't return anything useful (namely, a unit value).

Inside a computation expression, `do!` is very similar. Just as `let!` passes a wrapped result to the `Bind` method, so does `do!`, except that in the case of `do!` the "result" is the unit value, and so a *wrapped* version of unit is passed to the bind method.

Here is a simple demonstration using the `trace` workflow:

```
trace {
    do! Some (printfn "...expression that returns unit")
    do! Some (printfn "...another expression that returns unit")
    let! x = Some (1)
    return x
    } |> printfn "Result from do: %A"
```

Here is the output:

```
...expression that returns unit
Binding with Some(<null>). Continuing
...another expression that returns unit
Binding with Some(<null>). Continuing
Binding with Some(1). Continuing
Returning a unwrapped 1 as an option
Result from do: Some 1
```

You can verify for yourself that a `unit option` is being passed to `Bind` as a result of each `do!`.

Introducing "Zero"

What is the smallest computation expression you can get away with? Let's try nothing at all:

```
trace {
    } |> printfn "Result for empty: %A"
```

We get an error immediately:

```
This value is not a function and cannot be applied
```

Fair enough. If you think about it, it doesn't make sense to have nothing at all in a computation expression. After all, it's purpose is to chain expressions together.

Next, what about a simple expression with no `let!` or `return`?

```
trace {
    printfn "hello world"
    } |> printfn "Result for simple expression: %A"
```

Now we get a different error:

```
This control construct may only be used if the computation expression builder defines
a 'Zero' method
```

So why is the `Zero` method needed now but we haven't needed it before? The answer is that in this particular case we haven't returned anything explicitly, yet the computation expression as a whole *must* return a wrapped value. So what value should it return?

In fact, this situation will occur any time the return value of the computation expression has not been explicitly given. The same thing happens if you have an `if..then` expression without an else clause.

```
trace {
    if false then return 1
    } |> printfn "Result for if without else: %A"
```

In normal F# code, an "if..then" without an "else" would result in a unit value, but in a computation expression, the particular return value must be a member of the wrapped type, and the compiler does not know what value this is.

The fix is to tell the compiler what to use -- and that is the purpose of the `Zero` method.

What value should you use for Zero?

So which value *should* you use for `Zero`? It depends on the kind of workflow you are creating.

Here are some guidelines that might help:

- **Does the workflow have a concept of "success" or "failure"?** If so, use the "failure" value for `Zero`. For example, in our `trace` workflow, we use `None` to indicate failure,

and so we can use `None` as the Zero value.
- **Does the workflow have a concept of "sequential processing"?** That is, in your workflow you do one step and then another, with some processing behind the scenes. In normal F# code, an expression that did return anything explicitly would evaluate to unit. So to parallel this case, your `Zero` should be the *wrapped* version of unit. For example, in a variant on an option-based workflow, we might use `Some ()` to mean `Zero` (and by the way, this would always be the same as `Return ()` as well).
- **Is the workflow primarily concerned with manipulating data structures?** If so, `Zero` should be the "empty" data structure. For example, in a "list builder" workflow, we would use the empty list as the Zero value.

The `Zero` value also has an important role to play when combining wrapped types. So stay tuned, and we'll revisit Zero in the next post.

A Zero implementation

So now let's extend our testbed class with a `Zero` method that returns `None`, and try again.

```
type TraceBuilder() =
    // other members as before
    member this.Zero() =
        printfn "Zero"
        None

// make a new instance
let trace = new TraceBuilder()

// test
trace {
    printfn "hello world"
    } |> printfn "Result for simple expression: %A"

trace {
    if false then return 1
    } |> printfn "Result for if without else: %A"
```

The test code makes it clear that `Zero` is being called behind the scenes. And `None` is the return value for the expression as whole. *Note:* `None` *may print out as* `<null>`. *You can ignore this.*

Do you always need a Zero?

Remember, you *not required* to have a `Zero`, but only if it makes sense in the context of the workflow. For example `seq` does not allow zero, but `async` does:

```
let s = seq {printfn "zero" }    // Error
let a = async {printfn "zero" }  // OK
```

Introducing "Yield"

In C#, there is a "yield" statement that, within an iterator, is used to return early and then picks up where you left off when you come back.

And looking at the docs, there is a "yield" available in F# computation expressions as well. What does it do? Let's try it and see.

```
trace {
    yield 1
    } |> printfn "Result for yield: %A"
```

And we get the error:

```
This control construct may only be used if the computation expression builder defines
a 'Yield' method
```

No surprise there. So what should the implementation of "yield" method look like? The MSDN documentation says that it has the signature `'T -> M<'T>`, which is exactly the same as the signature for the `Return` method. It must take an unwrapped value and wrap it.

So let's implement it the same way as `Return` and retry the test expression.

```
type TraceBuilder() =
    // other members as before

    member this.Yield(x) =
        printfn "Yield an unwrapped %A as an option" x
        Some x

// make a new instance
let trace = new TraceBuilder()

// test
trace {
    yield 1
    } |> printfn "Result for yield: %A"
```

This works now, and it seems that it can be used as an exact substitute for `return`.

There is a also a `YieldFrom` method that parallels the `ReturnFrom` method. And it behaves the same way, allowing you to yield a wrapped value rather than a unwrapped one.

So let's add that to our list of builder methods as well:

```
type TraceBuilder() =
    // other members as before

    member this.YieldFrom(m) =
        printfn "Yield an option (%A) directly" m
        m

// make a new instance
let trace = new TraceBuilder()

// test
trace {
    yield! Some 1
    } |> printfn "Result for yield!: %A"
```

At this point you might be wondering: if `return` and `yield` are basically the same thing, why are there two different keywords? The answer is mainly so that you can enforce appropriate syntax by implementing one but not the other. For example, the `seq` expression *does* allow `yield` but *doesn't* allow `return`, while the `async` does allow `return`, but does not allow `yield`, as you can see from the snippets below.

```
let s = seq {yield 1}    // OK
let s = seq {return 1}   // error

let a = async {return 1} // OK
let a = async {yield 1}  // error
```

In fact, you could create slightly different behavior for `return` vs. `yield`, so that, for example, using `return` stops the rest of the computation expression from being evaluated, while `yield` doesn't.

More generally, of course, `yield` should be used for sequence/enumeration semantics, while `return` is normally used once per expression. (We'll see how `yield` can be used multiple times in the next post.)

Revisiting "For"

We talked about the `for..in..do` syntax in the last post. So now let's revisit the "list builder" that we discussed earlier and add the extra methods. We already saw how to define `Bind` and `Return` for a list in a previous post, so we just need to implement the additional methods.

- The `Zero` method just returns an empty list.
- The `Yield` method can be implemented in the same way as `Return`.
- The `For` method can be implemented the same as `Bind`.

```
type ListBuilder() =
    member this.Bind(m, f) =
        m |> List.collect f

    member this.Zero() =
        printfn "Zero"
        []

    member this.Return(x) =
        printfn "Return an unwrapped %A as a list" x
        [x]

    member this.Yield(x) =
        printfn "Yield an unwrapped %A as a list" x
        [x]

    member this.For(m,f) =
        printfn "For %A" m
        this.Bind(m,f)

// make an instance of the workflow
let listbuilder = new ListBuilder()
```

And here is the code using `let!`:

```
listbuilder {
    let! x = [1..3]
    let! y = [10;20;30]
    return x + y
    } |> printfn "Result: %A"
```

And here is the equivalent code using `for`:

```
listbuilder {
    for x in [1..3] do
    for y in [10;20;30] do
    return x + y
    } |> printfn "Result: %A"
```

You can see that both approaches give the same result.

Summary

In this post, we've seen how to implement the basic methods for a simple computation expression.

Some points to reiterate:

- For simple expressions you don't need to implement all the methods.
- Things with bangs have wrapped types on the right hand side.
- Things without bangs have unwrapped types on the right hand side.
- You need to implement `Zero` if you want a workflow that doesn't explicitly return a value.
- `Yield` is basically equivalent to `Return`, but `Yield` should be used for sequence/enumeration semantics.
- `For` is basically equivalent to `Bind` in simple cases.

In the next post, we'll look at what happens when we need to combine multiple values.

Implementing a builder: Combine

In this post we're going to look at returning multiple values from a computation expression using the `Combine` method.

The story so far...

So far, our expression builder class looks like this:

```
type TraceBuilder() =
    member this.Bind(m, f) =
        match m with
        | None ->
            printfn "Binding with None. Exiting."
        | Some a ->
            printfn "Binding with Some(%A). Continuing" a
        Option.bind f m

    member this.Return(x) =
        printfn "Returning a unwrapped %A as an option" x
        Some x

    member this.ReturnFrom(m) =
        printfn "Returning an option (%A) directly" m
        m

    member this.Zero() =
        printfn "Zero"
        None

    member this.Yield(x) =
        printfn "Yield an unwrapped %A as an option" x
        Some x

    member this.YieldFrom(m) =
        printfn "Yield an option (%A) directly" m
        m

// make an instance of the workflow
let trace = new TraceBuilder()
```

And this class has worked fine so far. But we are about to run into a problem...

A problem with two 'yields'

Previously, we saw how `yield` could be used to return values just like `return`.

Normally, `yield` is not used just once, of course, but multiple times in order to return values at different stages of a process such as an enumeration. So let's try that:

```
trace {
    yield 1
    yield 2
    } |> printfn "Result for yield then yield: %A"
```

But uh-oh, we get an error message:

```
This control construct may only be used if the computation expression builder defines
a 'Combine' method.
```

And if you use `return` instead of `yield`, you get the same error.

```
trace {
    return 1
    return 2
    } |> printfn "Result for return then return: %A"
```

And this problem occurs in other contexts too. For example, if we want to do something and then return, like this:

```
trace {
    if true then printfn "hello"
    return 1
    } |> printfn "Result for if then return: %A"
```

We get the same error message about a missing 'Combine' method.

Understanding the problem

So what's going on here?

To understand, let's go back to the behind-the-scenes view of the computation expression. We have seen that `return` and `yield` are really just the last step in a series of continuations, like this:

```
Bind(1,fun x ->
   Bind(2,fun y ->
     Bind(x + y,fun z ->
        Return(z)  // or Yield
```

You can think of `return` (or `yield`) as "resetting" the indentation, if you like. So when we `return/yield` and then `return/yield` again, we are generating code like this:

```
Bind(1,fun x ->
   Bind(2,fun y ->
     Bind(x + y,fun z ->
        Yield(z)
// start a new expression
Bind(3,fun w ->
   Bind(4,fun u ->
     Bind(w + u,fun v ->
        Yield(v)
```

But really this can be simplified to:

```
let value1 = some expression
let value2 = some other expression
```

In other words, we now have *two* values in our computation expression. And then the obvious question is, how should these two values be combined to give a single result for the computation expression as a whole?

This is a very important point. **Return and yield do *not* generate an early return from a computation expression**. No, the entire computation expression, all the way to the last curly brace, is *always* evaluated and results in a *single* value. Let me repeat that. Every part of the computation expression is *always evaluated* -- there is no short circuiting going on. If we want to short circuit and return early, we have to write our own code to do that (and we'll see how to do that later).

So, back to the pressing question. We have two expressions resulting in two values: how should those multiple values be combined into one?

Introducing "Combine"

The answer is by using the `Combine` method, which takes two *wrapped* values and combines them to make another wrapped value. Exactly how this works is up to us.

In our case, we are dealing specifically with `int options`, so one simple implementation that leaps to mind it just to add the numbers together. Each parameter is an `option` of course (the wrapped type), so we need to pick them apart and handle the four possible cases:

```
type TraceBuilder() =
    // other members as before

    member this.Combine (a,b) =
        match a,b with
        | Some a', Some b' ->
            printfn "combining %A and %A" a' b'
            Some (a' + b')
        | Some a', None ->
            printfn "combining %A with None" a'
            Some a'
        | None, Some b' ->
            printfn "combining None with %A" b'
            Some b'
        | None, None ->
            printfn "combining None with None"
            None

// make a new instance
let trace = new TraceBuilder()
```

Running the test code again:

```
trace {
    yield 1
    yield 2
    } |> printfn "Result for yield then yield: %A"
```

But now we get a different error message:

```
This control construct may only be used if the computation expression builder defines
a 'Delay' method
```

The `Delay` method is a hook that allows you to delay evaluation of a computation expression until needed -- we'll discuss this in detail very soon; but for now, let's create a default implementation:

```
type TraceBuilder() =
    // other members as before

    member this.Delay(f) =
        printfn "Delay"
        f()

// make a new instance
let trace = new TraceBuilder()
```

Running the test code again:

```
trace {
    yield 1
    yield 2
    } |> printfn "Result for yield then yield: %A"
```

And finally we get the code to complete.

```
Delay
Yield an unwrapped 1 as an option
Delay
Yield an unwrapped 2 as an option
combining 1 and 2
Result for yield then yield: Some 3
```

The result of the entire workflow is the sum of all the yields, namely `Some 3`.

If we have a "failure" in the workflow (e.g. a `None`), the second yield doesn't occur and the overall result is `Some 1` instead.

```
trace {
    yield 1
    let! x = None
    yield 2
    } |> printfn "Result for yield then None: %A"
```

We can have three `yields` rather than two:

```
trace {
    yield 1
    yield 2
    yield 3
    } |> printfn "Result for yield x 3: %A"
```

The result is what you would expect, `Some 6`.

We can even try mixing up `yield` and `return` together. Other than the syntax difference, the overall effect is the same.

```
trace {
    yield 1
    return 2
    } |> printfn "Result for yield then return: %A"

trace {
    return 1
    return 2
    } |> printfn "Result for return then return: %A"
```

Using Combine for sequence generation

Adding numbers up is not really the point of `yield`, although you might perhaps use a similar idea for constructing concatenated strings, somewhat like `StringBuilder`.

No, `yield` is naturally used as part of sequence generation, and now that we understand `Combine`, we can extend our "ListBuilder" workflow (from last time) with the required methods.

- The `Combine` method is just list concatenation.
- The `Delay` method can use a default implementation for now.

Here's the full class:

```
type ListBuilder() =
    member this.Bind(m, f) =
        m |> List.collect f

    member this.Zero() =
        printfn "Zero"
        []

    member this.Yield(x) =
        printfn "Yield an unwrapped %A as a list" x
        [x]

    member this.YieldFrom(m) =
        printfn "Yield a list (%A) directly" m
        m

    member this.For(m,f) =
        printfn "For %A" m
        this.Bind(m,f)

    member this.Combine (a,b) =
        printfn "combining %A and %A" a b
        List.concat [a;b]

    member this.Delay(f) =
        printfn "Delay"
        f()

// make an instance of the workflow
let listbuilder = new ListBuilder()
```

And here it is in use:

```
listbuilder {
    yield 1
    yield 2
    } |> printfn "Result for yield then yield: %A"

listbuilder {
    yield 1
    yield! [2;3]
    } |> printfn "Result for yield then yield! : %A"
```

And here's a more complicated example with a `for` loop and some `yield` s.

```
listbuilder {
    for i in ["red";"blue"] do
        yield i
        for j in ["hat";"tie"] do
            yield! [i + " " + j;"-"]
    } |> printfn "Result for for..in..do : %A"
```

And the result is:

```
["red"; "red hat"; "-"; "red tie"; "-"; "blue"; "blue hat"; "-"; "blue tie"; "-"]
```

You can see that by combining `for..in..do` with `yield`, we are not too far away from the built-in `seq` expression syntax (except that `seq` is lazy, of course).

I would strongly encourage you to play around with this a bit until you are clear on what is going on behind the scenes. As you can see from the example above, you can use `yield` in creative ways to generate all sorts of irregular lists, not just simple ones.

Note: If you're wondering about `While`, we're going to hold off on it for a bit, until after we have looked at `Delay` in an upcoming post.

Order of processing for "combine"

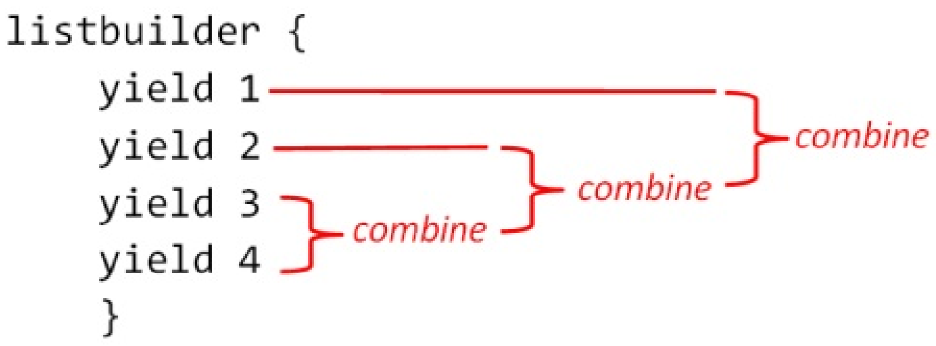
The `Combine` method only has two parameters. So what happens when you combine more than two values? For example, here are four values to combine:

```
listbuilder {
    yield 1
    yield 2
    yield 3
    yield 4
    } |> printfn "Result for yield x 4: %A"
```

If you look at the output you can see that the values are combined pair-wise, as you might expect.

```
combining [3] and [4]
combining [2] and [3; 4]
combining [1] and [2; 3; 4]
Result for yield x 4: [1; 2; 3; 4]
```

A subtle but important point is that they are combined "backwards", starting from the last value. First "3" is combined with "4", and the result of that is then combined with "2", and so on.



Combine for non-sequences

In the second of our earlier problematic examples, we didn't have a sequence; we just had two separate expressions in a row.

```
trace {
    if true then printfn "hello"  //expression 1
    return 1                      //expression 2
    } |> printfn "Result for combine: %A"
```

How should these expressions be combined?

There are a number of common ways of doing this, depending on the concepts that the workflow supports.

Implementing combine for workflows with "success" or "failure"

If the workflow has some concept of "success" or "failure", then a standard approach is:

- If the first expression "succeeds" (whatever that means in context), then use that value.
- Otherwise use the value of the second expression.

In this case, we also generally use the "failure" value for `Zero`.

This approach is useful for chaining together a series of "or else" expressions where the first success "wins" and becomes the overall result.

```
if (do first expression)
or else (do second expression)
or else (do third expression)
```

For example, for the `maybe` workflow, it is common to return the first expression if it is `Some`, but otherwise the second expression, like this:

```
type TraceBuilder() =
    // other members as before

    member this.Zero() =
        printfn "Zero"
        None  // failure

    member this.Combine (a,b) =
        printfn "Combining %A with %A" a b
        match a with
        | Some _ -> a  // a succeeds -- use it
        | None -> b    // a fails -- use b instead

// make a new instance
let trace = new TraceBuilder()
```

Example: Parsing

Let's try a parsing example with this implementation:

```
type IntOrBool = I of int | B of bool

let parseInt s =
    match System.Int32.TryParse(s) with
    | true,i -> Some (I i)
    | false,_ -> None

let parseBool s =
    match System.Boolean.TryParse(s) with
    | true,i -> Some (B i)
    | false,_ -> None

trace {
    return! parseBool "42"  // fails
    return! parseInt "42"
    } |> printfn "Result for parsing: %A"
```

We get the following result:

```
Some (I 42)
```

You can see that the first `return!` expression is `None`, and ignored. So the overall result is the second expression, `Some (I 42)`.

Example: Dictionary lookup

In this example, we'll try looking up the same key in a number of dictionaries, and return when we find a value:

```
let map1 = [ ("1","One"); ("2","Two") ] |> Map.ofList
let map2 = [ ("A","Alice"); ("B","Bob") ] |> Map.ofList

trace {
    return! map1.TryFind "A"
    return! map2.TryFind "A"
    } |> printfn "Result for map lookup: %A"
```

We get the following result:

```
Result for map lookup: Some "Alice"
```

You can see that the first lookup is `None`, and ignored. So the overall result is the second lookup.

As you can see, this technique is very convenient when doing parsing or evaluating a sequence of (possibly unsuccessful) operations.

Implementing combine for workflows with sequential steps

If the workflow has the concept of sequential steps, then the overall result is just the value of the last step, and all the previous steps are evaluated only for their side effects.

In normal F#, this would be written:

```
do some expression
do some other expression
final expression
```

Or using the semicolon syntax, just:

```
some expression; some other expression; final expression
```

In normal F#, each expression (other than the last) evaluates to the unit value.

The equivalent approach for a computation expression is to treat each expression (other than the last) as a *wrapped* unit value, and "pass it into" the next expression, and so on, until you reach the last expression.

This is exactly what bind does, of course, and so the easiest implementation is just to reuse the `Bind` method itself. Also, for this approach to work it is important that `Zero` is the wrapped unit value.

```
type TraceBuilder() =
    // other members as before

    member this.Zero() =
        printfn "Zero"
        this.Return ()  // unit not None

    member this.Combine (a,b) =
        printfn "Combining %A with %A" a b
        this.Bind( a, fun ()-> b )

// make a new instance
let trace = new TraceBuilder()
```

The difference from a normal bind is that the continuation has a unit parameter, and evaluates to `b`. This in turn forces `a` to be of type `WrapperType<unit>` in general, or `unit option` in our case.

Here's an example of sequential processing that works with this implementation of `Combine`:

```
trace {
    if true then printfn "hello......."
    if false then printfn ".......world"
    return 1
    } |> printfn "Result for sequential combine: %A"
```

Here's the following trace. Note that the result of the whole expression was the result of the last expression in the sequence, just like normal F# code.

```
hello.......
Zero
Returning a unwrapped <null> as an option
Zero
Returning a unwrapped <null> as an option
Returning a unwrapped 1 as an option
Combining Some null with Some 1
Combining Some null with Some 1
Result for sequential combine: Some 1
```

Implementing combine for workflows that build data structures

Finally, another common pattern for workflows is that they build data structures. In this case, `Combine` should merge the two data structures in whatever way is appropriate. And the `Zero` method should create an empty data structure, if needed (and if even possible).

In the "list builder" example above, we used exactly this approach. `Combine` was just list concatenation and `Zero` was the empty list.

Guidelines for mixing "Combine" and "Zero"

We have looked at two different implementations for `Combine` for option types.

- The first one used options as "success/failure" indicators, when the first success "won". In this case `Zero` was defined as `None`
- The second one was sequential, In this case `Zero` was defined as `Some ()`

Both cases worked nicely, but was that luck, or are there are any guidelines for implementing `Combine` and `Zero` correctly?

First, note that `Combine` does *not* have to give the same result if the parameters are swapped. That is, `Combine(a,b)` need not be the same as `Combine(b,a)`. The list builder is a good example of this.

On the other hand there is a useful rule that connects `Zero` and `Combine`.

Rule: `Combine(a,Zero)` should be the same as `Combine(Zero,a)` which should the same as just `a`.

To use an analogy from arithmetic, you can think of `Combine` like addition (which is not a bad analogy -- it really is "adding" two values). And `Zero` is just the number zero, of course! So the rule above can be expressed as:

Rule: `a + 0` is the same as `0 + a` is the same as just `a`, where `+` means `Combine` and `0` means `Zero`.

If you look at the first `Combine` implementation ("success/failure") for option types, you'll see that it does indeed comply with this rule, as does the second implementation ("bind" with `Some()`).

On the other hand, if we had used the "bind" implementation of `Combine` but left `Zero` defined as `None`, it would *not* have obeyed the addition rule, which would be a clue that we had got something wrong.

"Combine" without bind

As with all the builder methods, if you don't need them, you don't need to implement them. So for a workflow that is strongly sequential, you could easily create a builder class with `Combine`, `Zero`, and `Yield`, say, without having to implement `Bind` and `Return` at all.

Here's an example of a minimal implementation that works:

```
type TraceBuilder() =

    member this.ReturnFrom(x) = x

    member this.Zero() = Some ()

    member this.Combine (a,b) =
        a |> Option.bind (fun ()-> b )

    member this.Delay(f) = f()

// make an instance of the workflow
let trace = new TraceBuilder()
```

And here it is in use:

```
trace {
    if true then printfn "hello......."
    if false then printfn ".......world"
    return! Some 1
    } |> printfn "Result for minimal combine: %A"
```

Similarly, if you have a data-structure oriented workflow, you could just implement `Combine` and some other helpers. For example, here is a minimal implementation of our list builder class:

```
type ListBuilder() =

    member this.Yield(x) = [x]

    member this.For(m,f) =
        m |> List.collect f

    member this.Combine (a,b) =
        List.concat [a;b]

    member this.Delay(f) = f()

// make an instance of the workflow
let listbuilder = new ListBuilder()
```

And even with the minimal implementation, we can write code like this:

```
listbuilder {
    yield 1
    yield 2
    } |> printfn "Result: %A"

listbuilder {
    for i in [1..5] do yield i + 2
    yield 42
    } |> printfn "Result: %A"
```

A standalone "Combine" function

In a previous post, we saw that the "bind" function is often used as standalone function, and is normally given the operator `>>=`.

The `Combine` function too, is often used as a standalone function. Unlike bind, there is no standard symbol -- it can vary depending on how the combine function works.

A symmetric combination operation is often written as `++` or `<+>`. And the "left-biased" combination (that is, only do the second expression if the first one fails) that we used earlier for options is sometimes written as `<++`.

So here is an example of a standalone left-biased combination of options, as used in a dictionary lookup example.

```
module StandaloneCombine =

    let combine a b =
        match a with
        | Some _ -> a  // a succeeds -- use it
        | None -> b    // a fails -- use b instead

    // create an infix version
    let ( <++ ) = combine

    let map1 = [ ("1","One"); ("2","Two") ] |> Map.ofList
    let map2 = [ ("A","Alice"); ("B","Bob") ] |> Map.ofList

    let result =
        (map1.TryFind "A")
        <++ (map1.TryFind "B")
        <++ (map2.TryFind "A")
        <++ (map2.TryFind "B")
        |> printfn "Result of adding options is: %A"
```

Summary

What have we learned about `Combine` in this post?

- You need to implement `Combine` (and `Delay`) if you need to combine or "add" more than one wrapped value in a computation expression.
- `Combine` combines values pairwise, from last to first.
- There is no universal implementation of `Combine` that works in all cases -- it needs to be customized according the particular needs of the workflow.
- There is a sensible rule that relates `Combine` with `Zero`.
- `Combine` doesn't require `Bind` to be implemented.
- `Combine` can be exposed as a standalone function

In the next post, we'll add logic to control exactly when the internal expressions get evaluated, and introduce true short circuiting and lazy evaluation.

Implementing a builder: Delay and Run

In the last few posts we have covered all the basic methods (Bind, Return, Zero, and Combine) needed to create your own computation expression builder. In this post, we'll look at some of the extra features needed to make the workflow more efficient, by controlling when expressions get evaluated.

The problem: avoiding unnecessary evaluations

Let's say that we have created a "maybe" style workflow as before. But this time we want to use the "return" keyword to return early and stop any more processing being done.

Here is our complete builder class. The key method to look at is `Combine`, in which we simply ignore any secondary expressions after the first return.

```
type TraceBuilder() =
    member this.Bind(m, f) =
        match m with
        | None ->
            printfn "Binding with None. Exiting."
        | Some a ->
            printfn "Binding with Some(%A). Continuing" a
        Option.bind f m

    member this.Return(x) =
        printfn "Return an unwrapped %A as an option" x
        Some x

    member this.Zero() =
        printfn "Zero"
        None

    member this.Combine (a,b) =
        printfn "Returning early with %A. Ignoring second part: %A" a b
        a

    member this.Delay(f) =
        printfn "Delay"
        f()

// make an instance of the workflow
let trace = new TraceBuilder()
```

Let's see how it works by printing something, returning, and then printing something else:

```
trace {
    printfn "Part 1: about to return 1"
    return 1
    printfn "Part 2: after return has happened"
    } |> printfn "Result for Part1 without Part2: %A"
```

The debugging output should look something like the following, which I have annotated:

```
// first expression, up to "return"
Delay
Part 1: about to return 1
Return an unwrapped 1 as an option

// second expression, up to last curly brace.
Delay
Part 2: after return has happened
Zero   // zero here because no explicit return was given for this part

// combining the two expressions
Returning early with Some 1. Ignoring second part: <null>

// final result
Result for Part1 without Part2: Some 1
```

We can see a problem here. The "Part 2: after return" was printed, even though we were trying to return early.

Why? Well I'll repeat what I said in the last post: **return and yield do *not* generate an early return from a computation expression**. The entire computation expression, all the way to the last curly brace, is *always* evaluated and results in a single value.

This is a problem, because you might get unwanted side effects (such as printing a message in this case) and your code is doing something unnecessary, which might cause performance problems.

So, how can we avoid evaluating the second part until we need it?

Introducing "Delay"

The answer to the question is straightforward -- simply wrap part 2 of the expression in a function and only call this function when needed, like this.

```
let part2 =
    fun () ->
        printfn "Part 2: after return has happened"
        // do other stuff
        // return Zero

// only evaluate if needed
if needed then
   let result = part2()
```

Using this technique, part 2 of the computation expression can be processed completely, but because the expression returns a function, nothing actually *happens* until the function is called.
But the `Combine` method will never call it, and so the code inside it does not run at all.

And this is exactly what the `Delay` method is for. Any result from `Return` or `Yield` is immediately wrapped in a "delay" function like this, and then you can choose whether to run it or not.

Let's change the builder to implement a delay:

```
type TraceBuilder() =
    // other members as before

    member this.Delay(funcToDelay) =
        let delayed = fun () ->
            printfn "%A - Starting Delayed Fn." funcToDelay
            let delayedResult = funcToDelay()
            printfn "%A - Finished Delayed Fn. Result is %A" funcToDelay delayedResult
            delayedResult  // return the result

        printfn "%A - Delaying using %A" funcToDelay delayed
        delayed // return the new function
```

As you can see, the `Delay` method is given a function to execute. Previously, we executed it immediately. What we're doing now is wrapping this function in another function and returning the delayed function instead. I have added a number of trace statements before and after the function is wrapped.

If you compile this code, you can see that the signature of `Delay` has changed. Before the change, it returned a concrete value (an option in this case), but now it returns a function.

```
// signature BEFORE the change
member Delay : f:(unit -> 'a) -> 'a

// signature AFTER the change
member Delay : f:(unit -> 'b) -> (unit -> 'b)
```

By the way, we could have implemented `Delay` in a much simpler way, without any tracing, just by returning the same function that was passed in, like this:

```
member this.Delay(f) =
    f
```

Much more concise! But in this case, I wanted to add some detailed tracing information as well.

Now let's try again:

```
trace {
    printfn "Part 1: about to return 1"
    return 1
    printfn "Part 2: after return has happened"
    } |> printfn "Result for Part1 without Part2: %A"
```

Uh-oh. This time nothing happens at all! What went wrong?

If we look at the output we see this:

```
Result for Part1 without Part2: <fun:Delay@84-5>
```

Hmmm. The output of the whole `trace` expression is now a *function*, not an option. Why? Because we created all these delays, but we never "undelayed" them by actually calling the function!

One way to do this is to assign the output of the computation expression to a function value, say `f`, and then evaluate it.

```
let f = trace {
    printfn "Part 1: about to return 1"
    return 1
    printfn "Part 2: after return has happened"
    }
f() |> printfn "Result for Part1 without Part2: %A"
```

This works as expected, but is there a way to do this from inside the computation expression itself? Of course there is!

Introducing "Run"

The `Run` method exists for exactly this reason. It is called as the final step in the process of evaluating a computation expression, and can be used to undo the delay.

Here's an implementation:

```
type TraceBuilder() =
    // other members as before

    member this.Run(funcToRun) =
        printfn "%A - Run Start." funcToRun
        let runResult = funcToRun()
        printfn "%A - Run End. Result is %A" funcToRun runResult
        runResult // return the result of running the delayed function
```

Let's try one more time:

```
trace {
    printfn "Part 1: about to return 1"
    return 1
    printfn "Part 2: after return has happened"
    } |> printfn "Result for Part1 without Part2: %A"
```

And the result is exactly what we wanted. The first part is evaluated, but the second part is not. And the result of the entire computation expression is an option, not a function.

When is delay called?

The way that `Delay` is inserted into the workflow is straightforward, once you understand it.

- The bottom (or innermost) expression is delayed.
- If this is combined with a prior expression, the output of `Combine` is also delayed.
- And so on, until the final delay is fed into `Run`.

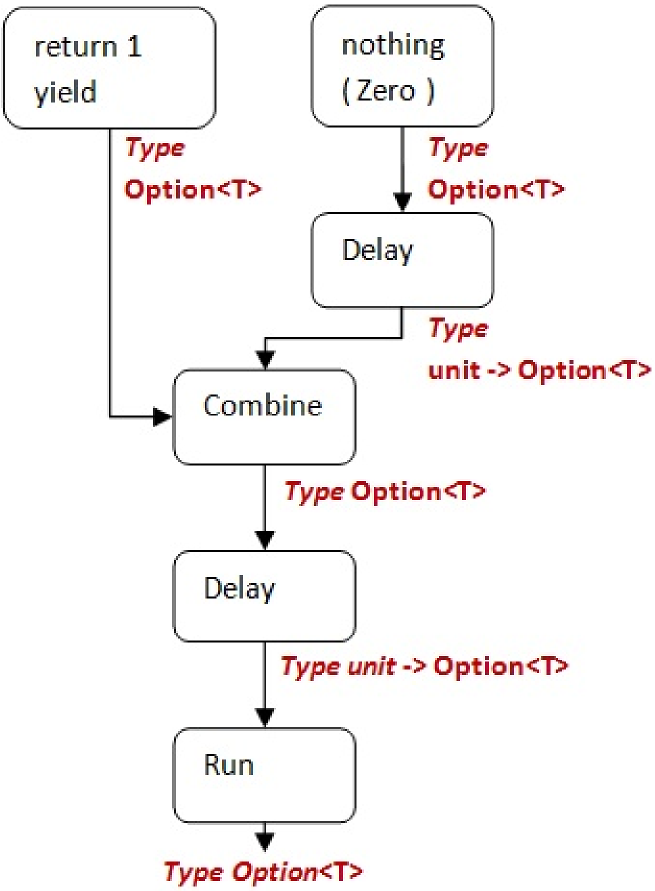
Using this knowledge, let's review what happened in the example above:

- The first part of the expression is the print statement plus `return 1`.
- The second part of the expression is the print statement without an explicit return, which means that `Zero()` is called
- The `None` from the `Zero` is fed into `Delay`, resulting in a "delayed option", that is, a function that will evaluate to an `option` when called.
- The option from part 1 and the delayed option from part 2 are combined in `Combine`

and the second one is discarded.

- The result of the combine is turned into another "delayed option".
- Finally, the delayed option is fed to `Run`, which evaluates it and returns a normal option.

Here is a diagram that represents this process visually:



If we look at the debug trace for the example above, we can see in detail what happened. It's a little confusing, so I have annotated it. Also, it helps to remember that working *down* this trace is the same as working *up* from the bottom of the diagram above, because the outermost code is run first.

```
// delaying the overall expression (the output of Combine)
<fun:clo@160-66> - Delaying using <fun:delayed@141-3>

// running the outermost delayed expression (the output of Combine)
<fun:delayed@141-3> - Run Start.
<fun:clo@160-66> - Starting Delayed Fn.

// the first expression results in Some(1)
Part 1: about to return 1
Return an unwrapped 1 as an option

// the second expression is wrapped in a delay
<fun:clo@162-67> - Delaying using <fun:delayed@141-3>

// the first and second expressions are combined
Combine. Returning early with Some 1. Ignoring <fun:delayed@141-3>

// overall delayed expression (the output of Combine) is complete
<fun:clo@160-66> - Finished Delayed Fn. Result is Some 1
<fun:delayed@141-3> - Run End. Result is Some 1

// the result is now an Option not a function
Result for Part1 without Part2: Some 1
```

"Delay" changes the signature of "Combine"

When `Delay` is introduced into the pipeline like this, it has an effect on the signature of `Combine`.

When we originally wrote `Combine` we were expecting it to handle `options`. But now it is handling the output of `Delay`, which is a function.

We can see this if we hard-code the types that `Combine` expects, with `int option` type annotations like this:

```
member this.Combine (a: int option,b: int option) =
    printfn "Returning early with %A. Ignoring %A" a b
    a
```

If this is done, we get an compiler error in the "return" expression:

```
trace {
    printfn "Part 1: about to return 1"
    return 1
    printfn "Part 2: after return has happened"
    } |> printfn "Result for Part1 without Part2: %A"
```

The error is:

```
error FS0001: This expression was expected to have type
    int option
but here has type
    unit -> 'a
```

In other words, the `Combine` is being passed a delayed function (`unit -> 'a`), which doesn't match our explicit signature.

So what happens when we *do* want to combine the parameters, but they are passed in as a function instead of as a simple value?

The answer is straightforward: just call the function that was passed in to get the underlying value.

Let's demonstrate that using the adding example from the previous post.

```
type TraceBuilder() =
    // other members as before

    member this.Combine (m,f) =
        printfn "Combine. Starting second param %A" f
        let y = f()
        printfn "Combine. Finished second param %A. Result is %A" f y

        match m,y with
        | Some a, Some b ->
            printfn "combining %A and %A" a b
            Some (a + b)
        | Some a, None ->
            printfn "combining %A with None" a
            Some a
        | None, Some b ->
            printfn "combining None with %A" b
            Some b
        | None, None ->
            printfn "combining None with None"
            None
```

In this new version of `Combine` , the *second* parameter is now a function, not an `int option` . So to combine them, we must first evaluate the function before doing the combination logic.

If we test this out:

```
trace {
    return 1
    return 2
    } |> printfn "Result for return then return: %A"
```

We get the following (annotated) trace:

```
// entire expression is delayed
<fun:clo@318-69> - Delaying using <fun:delayed@295-6>

// entire expression is run
<fun:delayed@295-6> - Run Start.

// delayed entire expression is run
<fun:clo@318-69> - Starting Delayed Fn.

// first return
Returning a unwrapped 1 as an option

// delaying second return
<fun:clo@319-70> - Delaying using <fun:delayed@295-6>

// combine starts
Combine. Starting second param <fun:delayed@295-6>

    // delayed second return is run inside Combine
    <fun:clo@319-70> - Starting Delayed Fn.
    Returning a unwrapped 2 as an option
    <fun:clo@319-70> - Finished Delayed Fn. Result is Some 2
    // delayed second return is complete

Combine. Finished second param <fun:delayed@295-6>. Result is Some 2
combining 1 and 2
// combine is complete

<fun:clo@318-69> - Finished Delayed Fn. Result is Some 3
// delayed entire expression is complete

<fun:delayed@295-6> - Run End. Result is Some 3
// Run is complete

// final result is printed
Result for return then return: Some 3
```

Understanding the type constraints

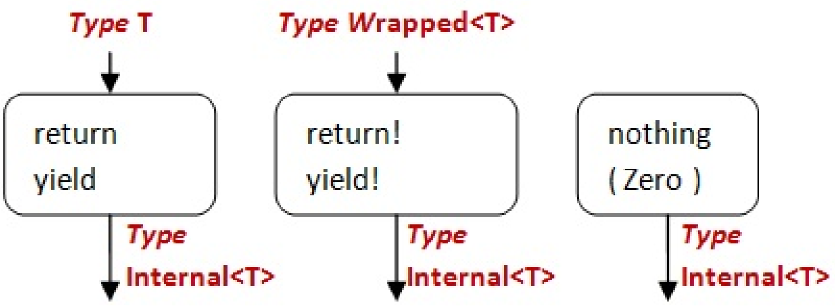
Up to now, we have used only our "wrapped type" (e.g. `int option`) and the delayed version (e.g. `unit -> int option`) in the implementation of our builder.

But in fact we can use other types if we like, subject to certain constraints. In fact, understanding exactly what the type constraints are in a computation expression can clarify how everything fits together.

For example, we have seen that:

- The output of `Return` is passed into `Delay`, so they must have compatible types.
- The output of `Delay` is passed into the second parameter of `Combine`.
- The output of `Delay` is also passed into `Run`.

But the output of `Return` does *not* have to be our "public" wrapped type. It could be an internally defined type instead.

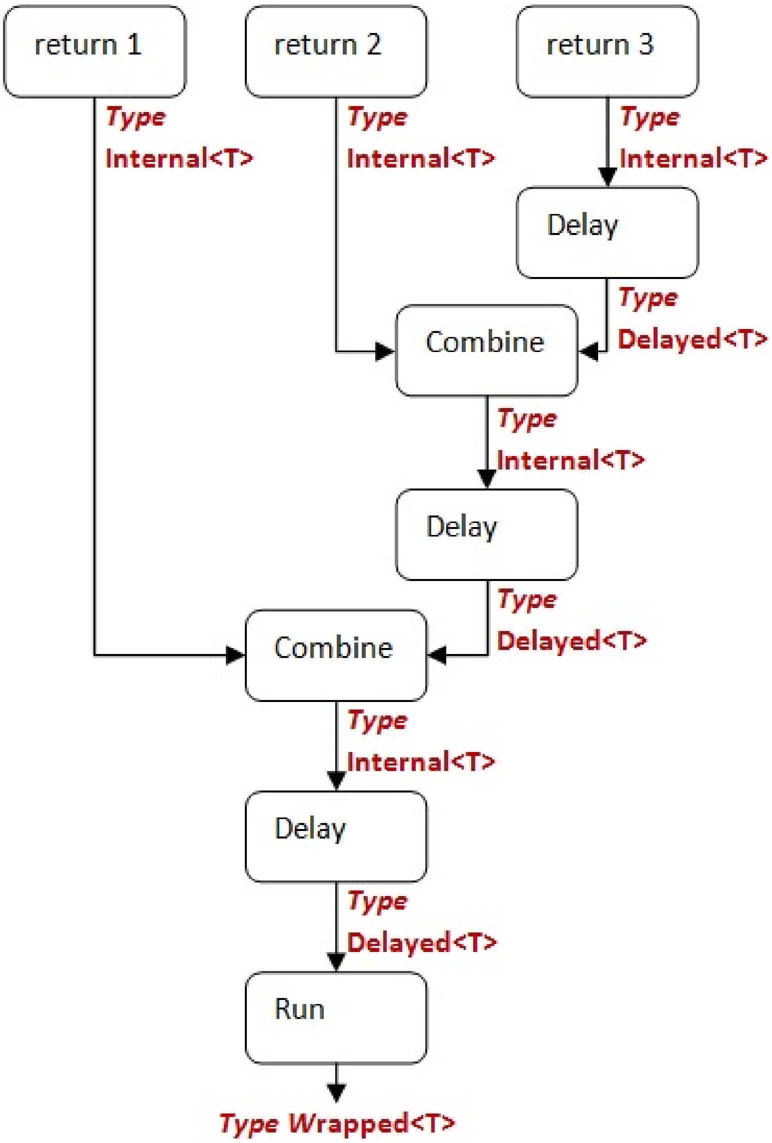


Similarly, the delayed type does not have to be a simple function, it could be any type that satisfies the constraints.

So, given a simple set of return expressions, like this:

```
trace {
    return 1
    return 2
    return 3
    } |> printfn "Result for return x 3: %A"
```

Then a diagram that represents the various types and their flow would look like this:



And to prove that this is valid, here is an implementation with distinct types for `Internal` and `Delayed` :

```
type Internal = Internal of int option
type Delayed = Delayed of (unit -> Internal)

type TraceBuilder() =
    member this.Bind(m, f) =
        match m with
        | None ->
            printfn "Binding with None. Exiting."
        | Some a ->
            printfn "Binding with Some(%A). Continuing" a
        Option.bind f m

    member this.Return(x) =
        printfn "Returning a unwrapped %A as an option" x
        Internal (Some x)
```

```
    member this.ReturnFrom(m) =
        printfn "Returning an option (%A) directly" m
        Internal m

    member this.Zero() =
        printfn "Zero"
        Internal None

    member this.Combine (Internal x, Delayed g) : Internal =
        printfn "Combine. Starting %A" g
        let (Internal y) = g()
        printfn "Combine. Finished %A. Result is %A" g y
        let o =
            match x,y with
            | Some a, Some b ->
                printfn "Combining %A and %A" a b
                Some (a + b)
            | Some a, None ->
                printfn "combining %A with None" a
                Some a
            | None, Some b ->
                printfn "combining None with %A" b
                Some b
            | None, None ->
                printfn "combining None with None"
                None
        // return the new value wrapped in a Internal
        Internal o

    member this.Delay(funcToDelay) =
        let delayed = fun () ->
            printfn "%A - Starting Delayed Fn." funcToDelay
            let delayedResult = funcToDelay()
            printfn "%A - Finished Delayed Fn. Result is %A" funcToDelay delayedResult
            delayedResult  // return the result

        printfn "%A - Delaying using %A" funcToDelay delayed
        Delayed delayed // return the new function wrapped in a Delay

    member this.Run(Delayed funcToRun) =
        printfn "%A - Run Start." funcToRun
        let (Internal runResult) = funcToRun()
        printfn "%A - Run End. Result is %A" funcToRun runResult
        runResult // return the result of running the delayed function

// make an instance of the workflow
let trace = new TraceBuilder()
```

And the method signatures in the builder class methods look like this:

```
type Internal = | Internal of int option
type Delayed = | Delayed of (unit -> Internal)

type TraceBuilder =
class
  new : unit -> TraceBuilder
  member Bind : m:'a option * f:('a -> 'b option) -> 'b option
  member Combine : Internal * Delayed -> Internal
  member Delay : funcToDelay:(unit -> Internal) -> Delayed
  member Return : x:int -> Internal
  member ReturnFrom : m:int option -> Internal
  member Run : Delayed -> int option
  member Zero : unit -> Internal
end
```

Creating this artifical builder is overkill of course, but the signatures clearly show how the various methods fit together.

Summary

In this post, we've seen that:

- You need to implement `Delay` and `Run` if you want to delay execution within a computation expression.
- Using `Delay` changes the signature of `Combine`.
- `Delay` and `Combine` can use internal types that are not exposed to clients of the computation expression.

The next logical step is wanting to delay execution *outside* a computation expression until you are ready, and that will be the topic on the next but one post. But first, we'll take a little detour to discuss method overloads.

Implementing a builder: Overloading

In this post, we'll take a detour and look at some tricks you can do with methods in a computation expression builder.

Ultimately, this detour will lead to a dead end, but I hope the journey might provide some more insight into good practices for designing your own computation expressions.

An insight: builder methods can be overloaded

At some point, you might have an insight:

- The builder methods are just normal class methods, and unlike standalone functions, methods can support overloading with different parameter types, which means we can create *different implementations* of any method, as long as the parameter types are different.

So then you might get excited about this and how it could be used. But it turns out to be less useful than you might think. Let's look at some examples.

Overloading "return"

Say that you have a union type. You might consider overloading `Return` or `Yield` with multiple implementations for each union case.

For example, here's a very simple example where `Return` has two overloads:

```
type SuccessOrError =
| Success of int
| Error of string

type SuccessOrErrorBuilder() =

    member this.Bind(m, f) =
        match m with
        | Success s -> f s
        | Error _ -> m

    /// overloaded to accept ints
    member this.Return(x:int) =
        printfn "Return a success %i" x
        Success x

    /// overloaded to accept strings
    member this.Return(x:string) =
        printfn "Return an error %s" x
        Error x

// make an instance of the workflow
let successOrError = new SuccessOrErrorBuilder()
```

And here it is in use:

```
successOrError {
    return 42
    } |> printfn "Result for success: %A"
// Result for success: Success 42

successOrError {
    return "error for step 1"
    } |> printfn "Result for error: %A"
//Result for error: Error "error for step 1"
```

What's wrong with this, you might think?

Well, first, if we go back to the discussion on wrapper types, we made the point that wrapper types should be *generic*. Workflows should be reusable as much as possible -- why tie the implementation to any particular primitive type?

What that means in this case is that the union type should be resigned to look like this:

```
type SuccessOrError<'a,'b> =
| Success of 'a
| Error of 'b
```

But as a consequence of the generics, the `Return` method can't be overloaded any more!

Second, it's probably not a good idea to expose the internals of the type inside the expression like this anyway. The concept of "success" and "failure" cases is useful, but a better way would be to hide the "failure" case and handle it automatically inside `Bind`, like this:

```
type SuccessOrError<'a,'b> =
| Success of 'a
| Error of 'b

type SuccessOrErrorBuilder() =

    member this.Bind(m, f) =
        match m with
        | Success s ->
            try
                f s
            with
            | e -> Error e.Message
        | Error _ -> m

    member this.Return(x) =
        Success x

// make an instance of the workflow
let successOrError = new SuccessOrErrorBuilder()
```

In this approach, `Return` is only used for success, and the failure cases are hidden.

```
successOrError {
    return 42
    } |> printfn "Result for success: %A"

successOrError {
    let! x = Success 1
    return x/0
    } |> printfn "Result for error: %A"
```

We'll see more of this technique in an upcoming post.

Multiple Combine implementations

Another time when you might be tempted to overload a method is when implementing `Combine`.

Let's revisit the `Combine` method for the `trace` workflow. If you remember, in the previous implementation of `Combine`, we just added the numbers together.

But what if we change our requirements, and say that:

- if we yield multiple values in the `trace` workflow, then we want to combine them into a list.

A first attempt using combine might look this:

```
member this.Combine (a,b) =
    match a,b with
    | Some a', Some b' ->
        printfn "combining %A and %A" a' b'
        Some [a';b']
    | Some a', None ->
        printfn "combining %A with None" a'
        Some [a']
    | None, Some b' ->
        printfn "combining None with %A" b'
        Some [b']
    | None, None ->
        printfn "combining None with None"
        None
```

In the `Combine` method, we unwrap the value from the passed-in option and combine them into a list wrapped in a `Some` (e.g. `Some [a';b']`).

For two yields it works as expected:

```
trace {
    yield 1
    yield 2
    } |> printfn "Result for yield then yield: %A"

// Result for yield then yield: Some [1; 2]
```

And for a yielding a `None`, it also works as expected:

```
trace {
    yield 1
    yield! None
    } |> printfn "Result for yield then None: %A"

// Result for yield then None: Some [1]
```

But what happens if there are *three* values to combine? Like this:

```
trace {
    yield 1
    yield 2
    yield 3
    } |> printfn "Result for yield x 3: %A"
```

If we try this, we get a compiler error:

```
error FS0001: Type mismatch. Expecting a
    int option
but given a
    'a list option
The type 'int' does not match the type ''a list'
```

What is the problem?

The answer is that after combining the 2nd and 3rd values (`yield 2; yield 3`), we get an option containing a *list of ints* or `int list option` . The error happens when we attempt to combine the first value (`Some 1`) with the combined value (`Some [2;3]`). That is, we are passing a `int list option` as the second parameter of `Combine` , but the first parameter is still a normal `int option` . The compiler is telling you that it wants the second parameter to be the same type as the first.

But, here's where we might want use our overloading trick. We can create *two* different implementations of `Combine` , with different types for the second parameter, one that takes an `int option` and the other taking an `int list option` .

So here are the two methods, with different parameter types:

```
/// combine with a list option
member this.Combine (a, listOption) =
    match a,listOption with
    | Some a', Some list ->
        printfn "combining %A and %A" a' list
        Some ([a'] @ list)
    | Some a', None ->
        printfn "combining %A with None" a'
        Some [a']
    | None, Some list ->
        printfn "combining None with %A" list
        Some list
    | None, None ->
        printfn "combining None with None"
        None

/// combine with a non-list option
member this.Combine (a,b) =
    match a,b with
    | Some a', Some b' ->
        printfn "combining %A and %A" a' b'
        Some [a';b']
    | Some a', None ->
        printfn "combining %A with None" a'
        Some [a']
    | None, Some b' ->
        printfn "combining None with %A" b'
        Some [b']
    | None, None ->
        printfn "combining None with None"
        None
```

Now if we try combining three results, as before, we get what we expect.

```
trace {
    yield 1
    yield 2
    yield 3
    } |> printfn "Result for yield x 3: %A"

// Result for yield x 3: Some [1; 2; 3]
```

Unfortunately, this trick has broken some previous code! If you try yielding a `None` now, you will get a compiler error.

```
trace {
    yield 1
    yield! None
    } |> printfn "Result for yield then None: %A"
```

The error is:

```
error FS0041: A unique overload for method 'Combine' could not be determined based on
type information prior to this program point. A type annotation may be needed.
```

But hold on, before you get too annoyed, try thinking like the compiler. If you were the compiler, and you were given a `None`, which method would *you* call?

There is no correct answer, because a `None` could be passed as the second parameter to *either* method. The compiler does not know where this is a None of type `int list option` (the first method) or a None of type `int option` (the second method).

As the compiler reminds us, a type annotation will help, so let's give it one. We'll force the None to be an `int option`.

```
trace {
    yield 1
    let x:int option = None
    yield! x
    } |> printfn "Result for yield then None: %A"
```

This is ugly, of course, but in practice might not happen very often.

More importantly, this is a clue that we have a bad design. Sometimes the computation expression returns an `'a option` and sometimes it returns an `'a list option`. We should be consistent in our design, so that the computation expression always returns the *same* type, no matter how many `yield`s are in it.

That is, if we *do* want to allow multiple `yield`s, then we should use `'a list option` as the wrapper type to begin with rather than just a plain option. In this case the `Yield` method would create the list option, and the `Combine` method could be collapsed to a single method again.

Here's the code for our third version:

```
type TraceBuilder() =
    member this.Bind(m, f) =
        match m with
        | None ->
            printfn "Binding with None. Exiting."
        | Some a ->
            printfn "Binding with Some(%A). Continuing" a
        Option.bind f m

    member this.Zero() =
        printfn "Zero"
        None

    member this.Yield(x) =
        printfn "Yield an unwrapped %A as a list option" x
        Some [x]

    member this.YieldFrom(m) =
        printfn "Yield an option (%A) directly" m
        m

    member this.Combine (a, b) =
        match a,b with
        | Some a', Some b' ->
            printfn "combining %A and %A" a' b'
            Some (a' @ b')
        | Some a', None ->
            printfn "combining %A with None" a'
            Some a'
        | None, Some b' ->
            printfn "combining None with %A" b'
            Some b'
        | None, None ->
            printfn "combining None with None"
            None

    member this.Delay(f) =
        printfn "Delay"
        f()

// make an instance of the workflow
let trace = new TraceBuilder()
```

And now the examples work as expected without any special tricks:

```
trace {
    yield 1
    yield 2
    } |> printfn "Result for yield then yield: %A"

// Result for yield then yield: Some [1; 2]

trace {
    yield 1
    yield 2
    yield 3
    } |> printfn "Result for yield x 3: %A"

// Result for yield x 3: Some [1; 2; 3]

trace {
    yield 1
    yield! None
    } |> printfn "Result for yield then None: %A"

// Result for yield then None: Some [1]
```

Not only is the code cleaner, but as in the `Return` example, we have made our code more generic as well, having gone from a specific type (`int option`) to a more generic type (`'a option`).

Overloading "For"

One legitimate case where overloading might be needed is the `For` method. Some possible reasons:

- You might want to support different kinds of collections (e.g. list *and* `IEnumerable`)
- You might have a more efficient looping implementation for certain kinds of collections.
- You might have a "wrapped" version of a list (e.g. LazyList) and you want support looping for both unwrapped and wrapped values.

Here's an example of our list builder that has been extended to support sequences as well as lists:

```
type ListBuilder() =
    member this.Bind(m, f) =
        m |> List.collect f

    member this.Yield(x) =
        printfn "Yield an unwrapped %A as a list" x
        [x]

    member this.For(m,f) =
        printfn "For %A" m
        this.Bind(m,f)

    member this.For(m:_ seq,f) =
        printfn "For %A using seq" m
        let m2 = List.ofSeq m
        this.Bind(m2,f)

// make an instance of the workflow
let listbuilder = new ListBuilder()
```

And here is it in use:

```
listbuilder {
    let list = [1..10]
    for i in list do yield i
    } |> printfn "Result for list: %A"

listbuilder {
    let s = seq {1..10}
    for i in s do yield i
    } |> printfn "Result for seq : %A"
```

If you comment out the second `For` method, you will see the "sequence` example will indeed fail to compile. So the overload is needed.

Summary

So we've seen that methods can be overloaded if needed, but be careful at jumping to this solution immediately, because having to doing this may be a sign of a weak design.

In the next post, we'll go back to controlling exactly when the expressions get evaluated, this time using a delay *outside* the builder.

Implementing a builder: Adding laziness

In a previous post, we saw how to avoid unnecessary evaluation of expressions in a workflow until needed.

But that approach was designed for expressions *inside* a workflow. What happens if we want to delay the *whole workflow itself* until needed.

The problem

Here is the code from our "maybe" builder class. This code is based on the `trace` builder from the earlier post, but with all the tracing taken out, so that it is nice and clean.

```
type MaybeBuilder() =

    member this.Bind(m, f) =
        Option.bind f m

    member this.Return(x) =
        Some x

    member this.ReturnFrom(x) =
        x

    member this.Zero() =
        None

    member this.Combine (a,b) =
        match a with
        | Some _ -> a  // if a is good, skip b
        | None -> b()  // if a is bad, run b

    member this.Delay(f) =
        f

    member this.Run(f) =
        f()

// make an instance of the workflow
let maybe = new MaybeBuilder()
```

Before moving on, make sure that you understand how this works. If we analyze this using the terminology of the earlier post, we can see that the types used are:

- Wrapper type: `'a option`
- Internal type: `'a option`
- Delayed type: `unit -> 'a option`

Now let's check this code and make sure everything works as expected.

```
maybe {
    printfn "Part 1: about to return 1"
    return 1
    printfn "Part 2: after return has happened"
    } |> printfn "Result for Part1 but not Part2: %A"

// result - second part is NOT evaluated

maybe {
    printfn "Part 1: about to return None"
    return! None
    printfn "Part 2: after None, keep going"
    } |> printfn "Result for Part1 and then Part2: %A"

// result - second part IS evaluated
```

But what happens if we refactor the code into a child workflow, like this:

```
let childWorkflow =
    maybe {printfn "Child workflow"}

maybe {
    printfn "Part 1: about to return 1"
    return 1
    return! childWorkflow
    } |> printfn "Result for Part1 but not childWorkflow: %A"
```

The output shows that the child workflow was evaluated even though it wasn't needed in the end. This might not be a problem in this case, but in many cases, we may not want this to happen.

So, how to avoid it?

Wrapping the inner type in a delay

The obvious approach is to wrap the *entire result of the builder* in a delay function, and then to "run" the result, we just evaluate the delay function.

So, here's our new wrapper type:

```
type Maybe<'a> = Maybe of (unit -> 'a option)
```

We've replaced a simple `option` with a function that evaluates to an option, and then wrapped that function in a single case union for good measure.

And now we need to change the `Run` method as well. Previously, it evaluated the delay function that was passed in to it, but now it should leave it unevaluated and wrap it in our new wrapper type:

```
// before
member this.Run(f) =
    f()

// after
member this.Run(f) =
    Maybe f
```

I've forgotten to fix up another method -- do you know which one? We'll bump into it soon!

One more thing -- we'll need a way to "run" the result now.

```
let run (Maybe f) = f()
```

Let's try out our new type on our previous examples:

```
let m1 = maybe {
    printfn "Part 1: about to return 1"
    return 1
    printfn "Part 2: after return has happened"
    }
```

Running this, we get something like this:

```
val m1 : Maybe<int> = Maybe <fun:m1@123-7>
```

That looks good; nothing else was printed.

And now run it:

```
run m1 |> printfn "Result for Part1 but not Part2: %A"
```

and we get the output:

```
Part 1: about to return 1
Result for Part1 but not Part2: Some 1
```

Perfect. Part 2 did not run.

But we run into a problem with the next example:

```
let m2 = maybe {
    printfn "Part 1: about to return None"
    return! None
    printfn "Part 2: after None, keep going"
    }
```

Oops! We forgot to fix up `ReturnFrom`! As we know, that method takes a *wrapped type*, and we have redefined the wrapped type now.

Here's the fix:

```
member this.ReturnFrom(Maybe f) =
    f()
```

We are going to accept a `Maybe` from outside, and then immediately run it to get at the option.

But now we have another problem -- we can't return an explicit `None` anymore in `return! None`, we have to return a `Maybe` type instead. How are we going to create one of these?

Well, we could create a helper function that constructs one for us. But there is a much simpler answer: you can create a new `Maybe` type by using a `maybe` expression!

```
let m2 = maybe {
    return! maybe {printfn "Part 1: about to return None"}
    printfn "Part 2: after None, keep going"
    }
```

This is why the `Zero` method is useful. With `Zero` and the builder instance, you can create new instances of the type even if they don't do anything.

But now we have one more error -- the dreaded "value restriction":

```
Value restriction. The value 'm2' has been inferred to have generic type
```

The reason why this has happened is that *both* expressions are returning `None`. But the compiler does not know what type `None` is. The code is using `None` of type `Option<obj>` (presumably because of implicit boxing) yet the compiler knows that the type can be more generic than that.

There are two fixes. One is to make the type explicit:

```
let m2_int: Maybe<int> = maybe {
    return! maybe {printfn "Part 1: about to return None"}
    printfn "Part 2: after None, keep going;"
    }
```

Or we can just return some non-None value instead:

```
let m2 = maybe {
    return! maybe {printfn "Part 1: about to return None"}
    printfn "Part 2: after None, keep going;"
    return 1
    }
```

Both of these solutions will fix the problem.

Now if we run the example, we see that the result is as expected. The second part *is* run this time.

```
run m2 |> printfn "Result for Part1 and then Part2: %A"
```

The trace output:

```
Part 1: about to return None
Part 2: after None, keep going;
Result for Part1 and then Part2: Some 1
```

Finally, we'll try the child workflow examples again:

```
let childWorkflow =
    maybe {printfn "Child workflow"}

let m3 = maybe {
    printfn "Part 1: about to return 1"
    return 1
    return! childWorkflow
    }

run m3 |> printfn "Result for Part1 but not childWorkflow: %A"
```

And now the child workflow is not evaluated, just as we wanted.

And if we *do* need the child workflow to be evaluated, this works too:

```
let m4 = maybe {
    return! maybe {printfn "Part 1: about to return None"}
    return! childWorkflow
    }

run m4 |> printfn "Result for Part1 and then childWorkflow: %A"
```

Reviewing the builder class

Let's look at all the code in the new builder class again:

```
type Maybe<'a> = Maybe of (unit -> 'a option)

type MaybeBuilder() =

    member this.Bind(m, f) =
        Option.bind f m

    member this.Return(x) =
        Some x

    member this.ReturnFrom(Maybe f) =
        f()

    member this.Zero() =
        None

    member this.Combine (a,b) =
        match a with
        | Some _' -> a    // if a is good, skip b
        | None -> b()     // if a is bad, run b

    member this.Delay(f) =
        f

    member this.Run(f) =
        Maybe f

// make an instance of the workflow
let maybe = new MaybeBuilder()

let run (Maybe f) = f()
```

If we analyze this new builder using the terminology of the earlier post, we can see that the types used are:

- Wrapper type: `Maybe<'a>`
- Internal type: `'a option`
- Delayed type: `unit -> 'a option`

Note that in this case it was convenient to use the standard `'a option` as the internal type, because we didn't need to modify `Bind` or `Return` at all.

An alternative design might use `Maybe<'a>` as the internal type as well, which would make things more consistent, but makes the code harder to read.

True laziness

Let's look at a variant of the last example:

```
let child_twice: Maybe<unit> = maybe {
    let workflow = maybe {printfn "Child workflow"}

    return! maybe {printfn "Part 1: about to return None"}
    return! workflow
    return! workflow
    }

run child_twice |> printfn "Result for childWorkflow twice: %A"
```

What should happen? How many times should the child workflow be run?

The delayed implementation above does ensure that the child workflow is only be evaluated on demand, but it does not stop it being run twice.

In some situations, you might require that the workflow is guaranteed to only run *at most once*, and then cached ("memoized"). This is easy enough to do using the `Lazy` type that is built into F#.

The changes we need to make are:

- Change `Maybe` to wrap a `Lazy` instead of a delay
- Change `ReturnFrom` and `run` to force the evaluation of the lazy value
- Change `Run` to run the delay from inside a `lazy`

Here is the new class with the changes:

```
type Maybe<'a> = Maybe of Lazy<'a option>

type MaybeBuilder() =

    member this.Bind(m, f) =
        Option.bind f m

    member this.Return(x) =
        Some x

    member this.ReturnFrom(Maybe f) =
        f.Force()

    member this.Zero() =
        None

    member this.Combine (a,b) =
        match a with
        | Some _' -> a    // if a is good, skip b
        | None -> b()     // if a is bad, run b

    member this.Delay(f) =
        f

    member this.Run(f) =
        Maybe (lazy f())

// make an instance of the workflow
let maybe = new MaybeBuilder()

let run (Maybe f) = f.Force()
```

And if we run the "child twice` code from above, we get:

```
Part 1: about to return None
Child workflow
Result for childWorkflow twice: <null>
```

from which it is clear that the child workflow only ran once.

Summary: Immediate vs. Delayed vs. Lazy

On this page, we've seen three different implementations of the `maybe` workflow. One that is always evaluated immediately, one that uses a delay function, and one that uses laziness with memoization.

So... which approach should you use?

There is no single "right" answer. Your choice depends on a number of things:

- *Is the code in the expression cheap to execute, and without important side-effects?* If so, stick with the first, immediate version. It's simple and easy to understand, and this is exactly what most implementations of the `maybe` workflow use.
- *Is the code in the expression expensive to execute, might the result vary with each call (e.g. non-deterministic), or are there important side-effects?* If so, use the second, delayed version. This is exactly what most other workflows do, especially those relating to I/O (such as `async`).
- F# does not attempt to be a purely functional language, so almost all F# code will fall into one of these two categories. But, *if you need to code in a guaranteed side-effect free style, or you just want to ensure that expensive code is evaluated at most once*, then use the third, lazy option.

Whatever your choice, do make it clear in the documentation. For example, the delayed vs. lazy implementations appear exactly the same to the client, but they have very different semantics, and the client code must be written differently for each case.

Now that we have finished with delays and laziness, we can go back to the builder methods and finish them off.

Implementing a builder: The rest of the standard methods

We're coming into the home stretch now. There are only a few more builder methods that need to be covered, and then you will be ready to tackle anything!

These methods are:

- `While` for repetition.
- `TryWith` and `TryFinally` for handling exceptions.
- `Use` for managing disposables

Remember, as always, that not all methods need to be implemented. If `While` is not relevant to you, don't bother with it.

One important note before we get started: **all the methods discussed here rely on delays** being used. If you are not using delay functions, then none of the methods will give the expected results.

Implementing "While"

We all know what "while" means in normal code, but what does it mean in the context of a computation expression? To understand, we have to revisit the concept of continuations again.

In previous posts, we saw that a series of expressions is converted into a chain of continuations like this:

```
Bind(1,fun x ->
   Bind(2,fun y ->
     Bind(x + y,fun z ->
        Return(z)  // or Yield
```

And this is the key to understanding a "while" loop -- it can be expanded in the same way.

First, some terminology. A while loop has two parts:

- There is a test at the top of the "while" loop which is evaluated each time to determine whether the body should be run. When it evaluates to false, the while loop is "exited". In computation expressions, the test part is known as the **"guard"**. The test function has no parameters, and returns a bool, so its signature is `unit -> bool`, of course.

- And there is the body of the "while" loop, evaluated each time until the "while" test fails. In computation expressions, this is a delay function that evaluates to a wrapped value. Since the body of the while loop is always the same, the same function is evaluated each time. The body function has no parameters, and returns nothing, and so its signature is just `unit -> wrapped unit`.

With this in place, we can create pseudo-code for a while loop using continuations:

```
// evaluate test function
let bool = guard()
if not bool
then
    // exit loop
    return what??
else
    // evaluate the body function
    body()

    // back to the top of the while loop

    // evaluate test function again
    let bool' = guard()
    if not bool'
    then
        // exit loop
        return what??
    else
        // evaluate the body function again
        body()

        // back to the top of the while loop

        // evaluate test function a third time
        let bool'' = guard()
        if not bool''
        then
            // exit loop
            return what??
        else
            // evaluate the body function a third time
            body()

            // etc
```

One question that is immediately apparent is: what should be returned when the while loop test fails? Well, we have seen this before with `if..then..`, and the answer is of course to use the `Zero` value.

The next thing is that the `body()` result is being discarded. Yes, it is a unit function, so there is no value to return, but even so, in our expressions, we want to be able to hook into this so we can add behavior behind the scenes. And of course, this calls for using the `Bind` function.

So here is a revised version of the pseudo-code, using `Zero` and `Bind` :

```
// evaluate test function
let bool = guard()
if not bool
then
    // exit loop
    return Zero
else
    // evaluate the body function
    Bind( body(), fun () ->

        // evaluate test function again
        let bool' = guard()
        if not bool'
        then
            // exit loop
            return Zero
        else
            // evaluate the body function again
            Bind( body(), fun () ->

                // evaluate test function a third time
                let bool'' = guard()
                if not bool''
                then
                    // exit loop
                    return Zero
                else
                    // evaluate the body function again
                    Bind( body(), fun () ->

                    // etc
```

In this case, the continuation function passed into `Bind` has a unit parameter, because the `body` function does not have a value.

Finally, the pseudo-code can be simplified by collapsing it into a recursive function like this:

```
member this.While(guard, body) =
    // evaluate test function
    if not (guard())
    then
        // exit loop
        this.Zero()
    else
        // evaluate the body function
        this.Bind( body(), fun () ->
            // call recursively
            this.While(guard, body))
```

And indeed, this is the standard "boiler-plate" implementation for `While` in almost all builder classes.

It is a subtle but important point that the value of `Zero` must be chosen properly. In previous posts, we saw that we could set the value for `Zero` to be `None` or `Some ()` depending on the workflow. For `While` to work however, the `Zero` *must be* set to `Some ()` and not `None`, because passing `None` into `Bind` will cause the whole thing to aborted early.

Also note that, although this is a recursive function, we didn't need the `rec` keyword. It is only needed for standalone functions that are recursive, not methods.

"While" in use

Let's look at it being used in the `trace` builder. Here's the complete builder class, with the `While` method:

```
type TraceBuilder() =
    member this.Bind(m, f) =
        match m with
        | None ->
            printfn "Binding with None. Exiting."
        | Some a ->
            printfn "Binding with Some(%A). Continuing" a
        Option.bind f m

    member this.Return(x) =
        Some x

    member this.ReturnFrom(x) =
        x

    member this.Zero() =
        printfn "Zero"
        this.Return ()

    member this.Delay(f) =
        printfn "Delay"
        f

    member this.Run(f) =
        f()

    member this.While(guard, body) =
        printfn "While: test"
        if not (guard())
        then
            printfn "While: zero"
            this.Zero()
        else
            printfn "While: body"
            this.Bind( body(), fun () ->
                this.While(guard, body))

// make an instance of the workflow
let trace = new TraceBuilder()
```

If you look at the signature for `While`, you will see that the `body` parameter is `unit -> unit option`, that is, a delayed function. As noted above, if you don't implement `Delay` properly, you will get unexpected behavior and cryptic compiler errors.

```
type TraceBuilder =
    // other members
    member
      While : guard:(unit -> bool) * body:(unit -> unit option) -> unit option
```

And here is a simple loop using a mutable value that is incremented each time round.

```
let mutable i = 1
let test() = i < 5
let inc() = i <- i + 1

let m = trace {
    while test() do
        printfn "i is %i" i
        inc()
    }
```

Handling exceptions with "try..with"

Exception handling is implemented in a similar way.

If we look at a `try..with` expression for example, it has two parts:

- There is the body of the "try", evaluated once. In a computation expressions, this will be a delayed function that evaluates to a wrapped value. The body function has no parameters, and so its signature is just `unit -> wrapped type`.
- The "with" part handles the exception. It has an exception as a parameters, and returns the same type as the "try" part, so its signature is `exception -> wrapped type`.

With this in place, we can create pseudo-code for the exception handler:

```
try
    let wrapped = delayedBody()
    wrapped  // return a wrapped value
with
| e -> handlerPart e
```

And this maps exactly to a standard implementation:

```
member this.TryWith(body, handler) =
    try
        printfn "TryWith Body"
        this.ReturnFrom(body())
    with
        e ->
            printfn "TryWith Exception handling"
            handler e
```

As you can see, it is common to use pass the returned value through `ReturnFrom` so that it gets the same treatment as other wrapped values.

Here is an example snippet to test how the handling works:

```
trace {
    try
        failwith "bang"
    with
    | e -> printfn "Exception! %s" e.Message
    } |> printfn "Result %A"
```

Implementing "try..finally"

`try..finally` is very similar to `try..with`.

- There is the body of the "try", evaluated once. The body function has no parameters, and so its signature is `unit -> wrapped type`.
- The "finally" part is always called. It has no parameters, and returns a unit, so its signature is `unit -> unit`.

Just as with `try..with`, the standard implementation is obvious.

```
member this.TryFinally(body, compensation) =
    try
        printfn "TryFinally Body"
        this.ReturnFrom(body())
    finally
        printfn "TryFinally compensation"
        compensation()
```

Another little snippet:

```
trace {
    try
        failwith "bang"
    finally
        printfn "ok"
    } |> printfn "Result %A"
```

Implementing "using"

The final method to implement is `Using`. This is the builder method for implementing the `use!` keyword.

This is what the MSDN documentation says about `use!`:

```
{| use! value = expr in cexpr |}
```

is translated to:

```
builder.Bind(expr, (fun value -> builder.Using(value, (fun value -> {| cexpr |} ))))
```

In other words, the `use!` keyword triggers both a `Bind` and a `Using`. First a `Bind` is done to unpack the wrapped value, and then the unwrapped disposable is passed into `Using` to ensure disposal, with the continuation function as the second parameter.

Implementing this is straightforward. Similar to the other methods, we have a body, or continuation part, of the "using" expression, which is evaluated once. This body function has a "disposable" parameter, and so its signature is `#IDisposable -> wrapped type`.

Of course we want to ensure that the disposable value is always disposed no matter what, so we need to wrap the call to the body function in a `TryFinally`.

Here's a standard implementation:

```
member this.Using(disposable:#System.IDisposable, body) =
    let body' = fun () -> body disposable
    this.TryFinally(body', fun () ->
        match disposable with
            | null -> ()
            | disp -> disp.Dispose())
```

Notes:

- The parameter to `TryFinally` is a `unit -> wrapped`, with a *unit* as the first parameter, so we created a delayed version of the body that is passed in.
- Disposable is a class, so it could be `null`, and we have to handle that case specially. Otherwise we just dispose it in the "finally" continuation.

Here's a demonstration of `Using` in action. Note that the `makeResource` makes a *wrapped* disposable. If it wasn't wrapped, we wouldn't need the special `use!` and could just use a normal `use` instead.

```
let makeResource name =
    Some {
    new System.IDisposable with
    member this.Dispose() = printfn "Disposing %s" name
    }

trace {
    use! x = makeResource "hello"
    printfn "Disposable in use"
    return 1
    } |> printfn "Result: %A"
```

"For" revisited

Finally, we can revisit how `For` is implemented. In the previous examples, `For` took a simple list parameter. But with `Using` and `While` under our belts, we can change it to accept any `IEnumerable<_>` or sequence.

Here's the standard implementation for `For` now:

```
member this.For(sequence:seq<_>, body) =
       this.Using(sequence.GetEnumerator(),fun enum ->
            this.While(enum.MoveNext,
                this.Delay(fun () -> body enum.Current)))
 {% endhighlight fsharp %}

As you can see, it is quite different from the previous implementation, in order to ha
ndle a generic `IEnumerable<_>`.

* We explicitly iterate using an `IEnumerator<_>`.
* `IEnumerator<_>` implements `IDisposable`, so we wrap the enumerator in a `Using`.
* We use `While .. MoveNext` to iterate.
* Next, we pass the `enum.Current` into the body function
* Finally, we delay the call to the body function using `Delay`

## Complete code without tracing

Up to now, all the builder methods have been made more complex than necessary by the a
dding of tracing and printing expressions. The tracing is helpful to understand what i
s going on,
but it can obscure the simplicity of the methods.

So as a final step, let's have a look at the complete code for the "trace" builder cla
ss, but this time without any extraneous code at all.  Even though the code is cryptic
, the purpose and implementation of each method should now be familiar to you.

```fsharp
type TraceBuilder() =
```
```

```
    member this.Bind(m, f) =
        Option.bind f m

    member this.Return(x) = Some x

    member this.ReturnFrom(x) = x

    member this.Yield(x) = Some x

    member this.YieldFrom(x) = x

    member this.Zero() = this.Return ()

    member this.Delay(f) = f

    member this.Run(f) = f()

    member this.While(guard, body) =
        if not (guard())
        then this.Zero()
        else this.Bind( body(), fun () ->
            this.While(guard, body))

    member this.TryWith(body, handler) =
        try this.ReturnFrom(body())
        with e -> handler e

    member this.TryFinally(body, compensation) =
        try this.ReturnFrom(body())
        finally compensation()

    member this.Using(disposable:#System.IDisposable, body) =
        let body' = fun () -> body disposable
        this.TryFinally(body', fun () ->
            match disposable with
                | null -> ()
                | disp -> disp.Dispose())

    member this.For(sequence:seq<_>, body) =
        this.Using(sequence.GetEnumerator(),fun enum ->
            this.While(enum.MoveNext,
                this.Delay(fun () -> body enum.Current)))
```

After all this discussion, the code seems quite tiny now. And yet this builder implements every standard method, uses delayed functions. A lot of functionality in a just a few lines!

Organizing modules in a project

Before we move on to any coding in the recipe, let's look at the overall structure of a F# project. In particular: (a) what code should be in which modules and (b) how the modules should be organized within a project.

How not to do it

A newcomer to F# might be tempted to organize code in classes just like in C#. One class per file, in alphabetical order. After all, F# supports the same object-oriented features that C# does, right? So surely the F# code can be organized the same way as C# code?

After a while, this is often followed by the discovery that F# requires files (and code within a file) to be in *dependency order*. That is, you cannot use forward references to code that hasn't been seen by the compiler yet**.

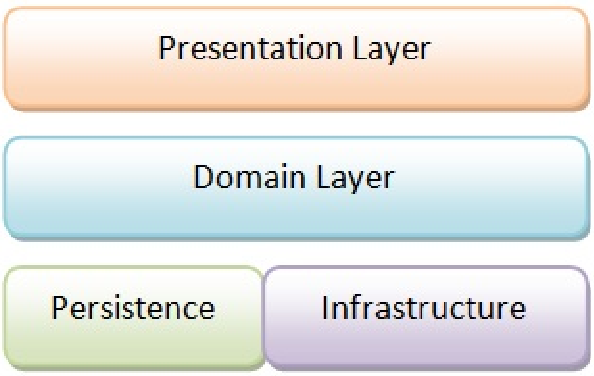
This is followed by general annoyance and swearing. How can F# be so stupid? Surely it impossible to write any kind of large project!

In this post, we'll look at one simple way to organize your code so that this doesn't happen.

** The `and` keyword can be used in some cases to allow mutual recursion, but is discouraged.

The functional approach to layered design

A standard way of thinking about code is to group it into layers: a domain layer, a presentation layer, and so on, like this:

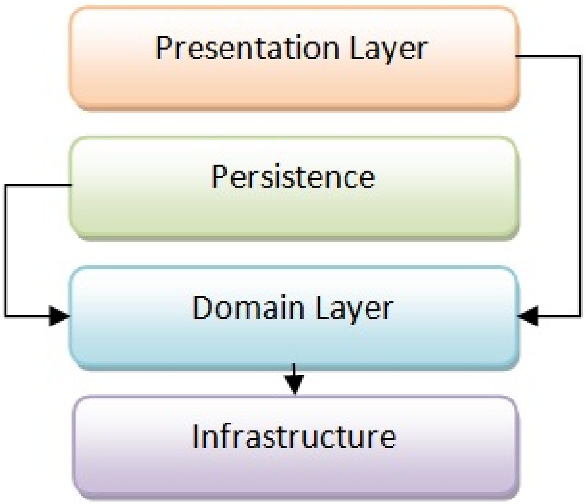


Each layer contains *only* the code that is relevant to that layer.

But in practice, it is not that simple, as there are dependencies between each layer. The domain layer depends on the infrastructure, and the presentation layer depends on the domain.

And most importantly, the domain layer should *not* depend on the persistence layer. That is, it should be "persistence agnostic".

We therefore need to tweak the layer diagram to look more like this (where each arrow represents a dependency):



And ideally this reorganization would be made even more fine grained, with a separate "Service Layer", containing application services, domain services, etc. And when we are finished, the core domain classes are "pure" and have no dependencies on anything else outside the domain. This is often called a "hexagonal architecture" or "onion architecture". But this post is not about the subtleties of OO design, so for now, let's just work with the simpler model.

Separating behavior from types

"It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures" -- Alan Perlis

In a functional design, it is very important to *separate behavior from data*. The data types are simple and "dumb". And then separately, you have a number of functions that act on those data types.

This is the exact opposite of an object-oriented design, where behavior and data are meant to be combined. After all, that's exactly what a class is. In a truly object-oriented design in fact, you should have nothing *but* behavior -- the data is private and can only be accessed via methods.

In fact, in OOD, not having enough behavior around a data type is considered a Bad Thing, and even has a name: the "anemic domain model".

In functional design though, having "dumb data" with transparency is preferred. It is normally fine for the data to be exposed without being encapsulated. The data is immutable, so it can't get "damaged" by a misbehaving function. And it turns out that the focus on transparent data allows for more code that is more flexible and generic.

If you haven't seen it, I highly recommend Rich Hickey's excellent talk on "The Value of Values", which explains the benefits of this approach.

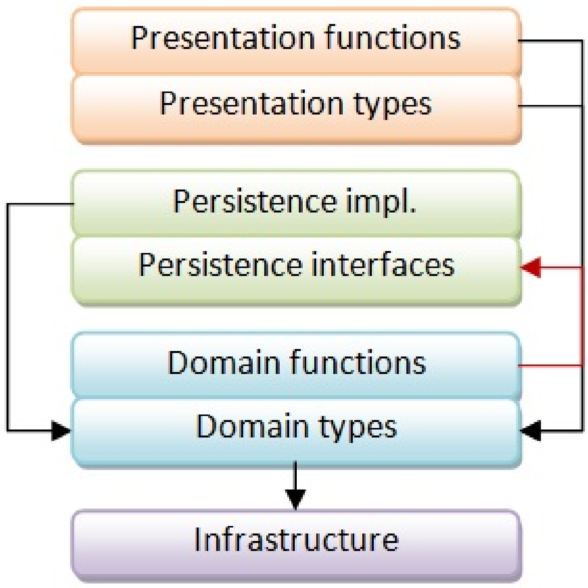
Type layers and behavior layers

So how does this apply to our layered design from above?

First, we must separate each layer into two distinct parts:

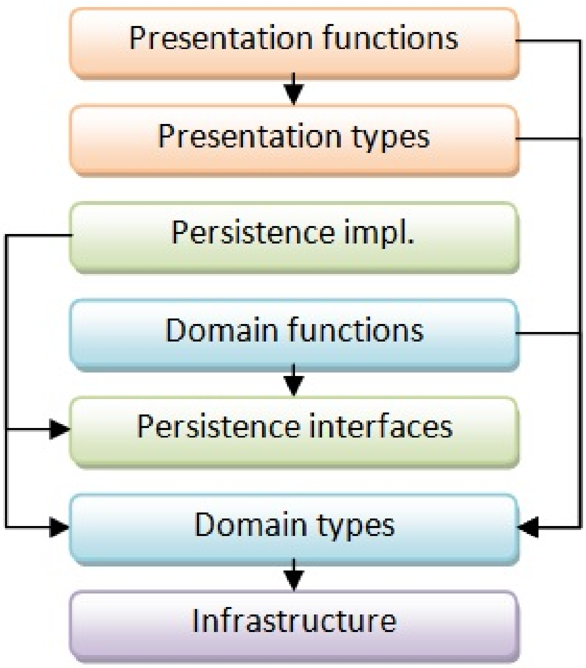
- **Data Types**. Data structures that are used by that layer.
- **Logic**. Functions that are implemented in that layer.

Once we have separated these two elements, our diagram will look like this:



Notice though, that we might have some backwards references (shown by the red arrow). For example, a function in the domain layer might depend on a persistence-related type, such as `IRepository`.

In an OO design, we would add more layers (e.g. application services) to handle this. But in a functional design, we don't need to -- we can just move the persistence-related types to a different place in the hierarchy, underneath the domain functions, like this:



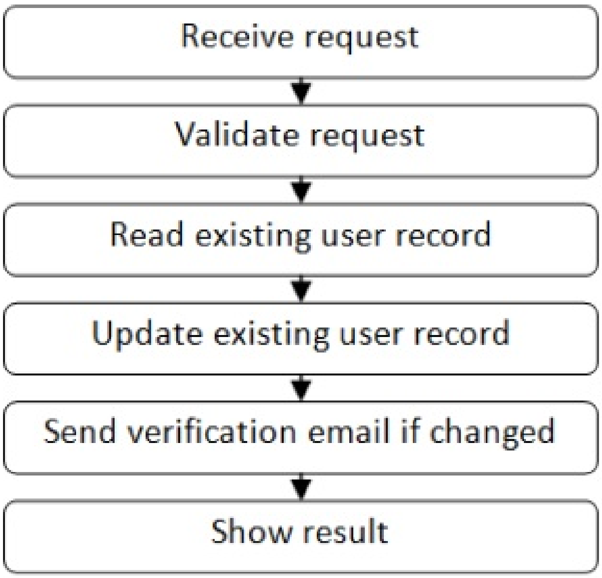
In this design, we have now eliminated all cyclic references between layers. *All the arrows point down*.

And this without having to create any extra layers or overhead.

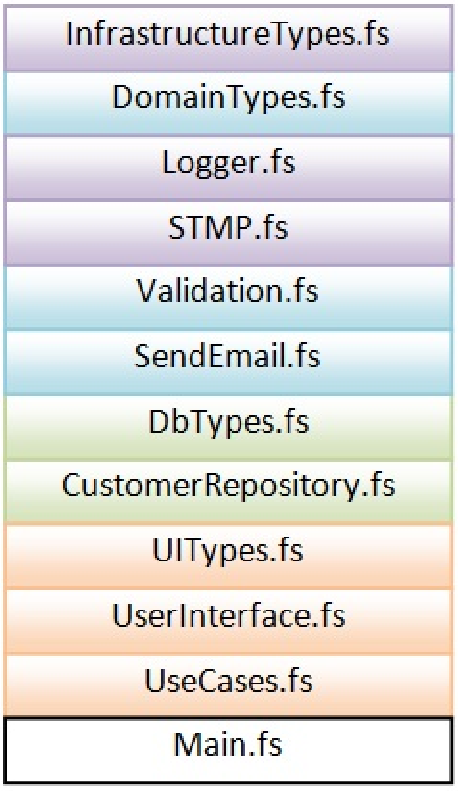
Finally, we can translate this layered design into F# files by turning it upside down.

- The first file in the project should contain code which has no dependencies. This represents the functionality at the *bottom* of the layer diagram. It is generally a set of types, such the infrastructure or domain types.
- The next file depends only on the first file. It would represents the functionality at the next-to-bottom layer.
- And so on. Each file depends only on the previous ones.

So, if we refer back to the use case example discussed in Part 1:



then the corresponding code in an F# project might look something like this:



At the very bottom of the list is the main file, called "main" or "program", which contains the entry point for the program.

And just above it is the code for the use cases in the application. The code in this file is where all the functions from all the other modules are "glued together" into a single function that represents a particular use case or service request. (The nearest equivalent of this in an OO design are the "application services", which serve roughly the same purpose.)

And then just above that is the "UI layer" and then the "DB layer" and so on, until you get to the top.

What's nice about this approach is that, if you are a newcomer to a code base, you always know where to start. The first few files will always be the "bottom layer" of an application and the last few files will always be the "top layer". No folders needed!

Putting code in modules, not classes

A common question from newcomers to F# is "how should I organize my code if I don't use classes?"

The answer is: *modules*. As you know, in an object oriented program, a data structure and the functions that act on it would be combined in a class. However in functional-style F#, a data structure and the functions that act on it are contained in modules instead.

There are three common patterns for mixing types and functions together:

- having the type declared in the same module as the functions.
- having the type declared separately from the functions but in the same file.
- having the type declared separately from the functions and in a different file, typically containing type definitions only.

In the first approach, types are defined *inside* the module along with their related functions. If there is only one primary type, it is often given a simple name such as "T" or the name of the module.

Here's an example:

```
namespace Example

// declare a module
module Person =

    type T = {First:string; Last:string}

    // constructor
    let create first last =
        {First=first; Last=last}

    // method that works on the type
    let fullName {First=first; Last=last} =
        first + " " + last
```

So the functions are accessed with names like `Person.create` and `Person.fullName` while the type itself is accessed with the name `Person.T`.

In the second approach, types are declared in the same file, but outside any module:

```
namespace Example

// declare the type outside the module
type PersonType = {First:string; Last:string}

// declare a module for functions that work on the type
module Person =

    // constructor
    let create first last =
        {First=first; Last=last}

    // method that works on the type
    let fullName {First=first; Last=last} =
        first + " " + last
```

In this case, the functions are accessed with the same names (`Person.create` and `Person.fullName`) while the type itself is accessed with the name such as `PersonType`.

And finally, here's the third approach. The type is declared in a special "types-only" module (typically in a different file):

```
// =========================
// File: DomainTypes.fs
// =========================
namespace Example

// "types-only" module
[<AutoOpen>]
module DomainTypes =

    type Person = {First:string; Last:string}

    type OtherDomainType = ...

    type ThirdDomainType = ...
```

In this particular case, the `AutoOpen` attribute has been used to make the types in this module automatically visible to all the other modules in the project -- making them "global".

And then a different module contains all the functions that work on, say, the `Person` type.

```
// ==========================
// File: Person.fs
// ==========================
namespace Example

// declare a module for functions that work on the type
module Person =

    // constructor
    let create first last =
        {First=first; Last=last}

    // method that works on the type
    let fullName {First=first; Last=last} =
        first + " " + last
```

Note that in this example, both the type and the module are called `Person`. This is not normally a problem in practice, as the compiler can normally figure out what you want.

So, if you write this:

```
let f (p:Person) = p.First
```

Then the compiler will understand that you are referring to the `Person` type.

On the other hand, if you write this:

```
let g () = Person.create "Alice" "Smith"
```

Then the compiler will understand that you are referring to the `Person` module.

For more on modules, see the post on organizing functions.

The organization of the modules

For our recipe we will use a mixture of approaches, with the following guidelines:

Module Guidelines

If a type is shared among multiple modules, then put it in a special types-only module.

- For example, if a type is used globally (or to be precise, within a "bounded domain" in DDD-speak), I would put it in a module called `DomainTypes` or `DomainModel`, which comes early in the compilation order.
- If a type is used only in a subsystem, such as a type shared by a number of UI modules,

then I would put it in a module called `UITypes`, which would come just before the other UI modules in the compilation order.

If a type is private to a module (or two) then put it in the same module as its related functions.

- For example, a type that was used only for validation would be put in the `Validation` module. A type used only for database access would be put in the `Database` module, and so on.

Of course, there are many ways to organize types, but these guidelines act as a good default starting point.

Dude, where are my folders?

A common complaint is that F# projects do not support a folder structure, which supposedly makes it hard to organize large projects.

If you are doing a pure object-oriented design, then this is a legitimate complaint. But as you can see from the discussion above, having a linear list of modules is very helpful (if not strictly necessary) to ensure that the dependencies are maintained correctly. Yes, in theory, the files could be scattered about and the compiler might be able to figure out the correct compilation order, but in practice, it's not easy for the compiler to determine this order.

Even more importantly, it's not easy for a *human* to determine the correct order either, and so it would make maintenance more painful than it needed to be.

In reality, even for large projects, not having folders is not as much of a problem as you might think. There are a number of large F# projects which successfully work within this limitation, such as the F# compiler itself. See the post on cycles and modularity in the wild for more.

Help, I have mutual dependencies between my types

If you are coming from an OO design, you might run into mutual dependencies between types, such as this example, which won't compile:

```
type Location = {name: string; workers: Employee list}

type Employee = {name: string; worksAt: Location}
```

How can you fix this to make the F# compiler happy?

It's not that hard, but it does requires some more explanation, so I have devoted another whole post to dealing with cyclic dependencies.

Example code

Let's revisit at the code we have so far, but this time organized into modules.

Each module below would typically become a separate file.

Be aware that this is still a skeleton. Some of the modules are missing, and some of the modules are almost empty.

This kind of organization would be overkill for a small project, but there will be lots more code to come!

```
/// ===========================================
/// Common types and functions shared across multiple projects
/// ===========================================
module CommonLibrary =

    // the two-track type
    type Result<'TSuccess,'TFailure> =
        | Success of 'TSuccess
        | Failure of 'TFailure

    // convert a single value into a two-track result
    let succeed x =
        Success x

    // convert a single value into a two-track result
    let fail x =
        Failure x

    // appy either a success function or failure function
    let either successFunc failureFunc twoTrackInput =
        match twoTrackInput with
        | Success s -> successFunc s
        | Failure f -> failureFunc f


    // convert a switch function into a two-track function
    let bind f =
        either f fail

    // pipe a two-track value into a switch function
    let (>>=) x f =
        bind f x

    // compose two switches into another switch
```

```
    let (>=>) s1 s2 =
        s1 >> bind s2

    // convert a one-track function into a switch
    let switch f =
        f >> succeed

    // convert a one-track function into a two-track function
    let map f =
        either (f >> succeed) fail

    // convert a dead-end function into a one-track function
    let tee f x =
        f x; x

    // convert a one-track function into a switch with exception handling
    let tryCatch f exnHandler x =
        try
            f x |> succeed
        with
        | ex -> exnHandler ex |> fail

    // convert two one-track functions into a two-track function
    let doubleMap successFunc failureFunc =
        either (successFunc >> succeed) (failureFunc >> fail)

    // add two switches in parallel
    let plus addSuccess addFailure switch1 switch2 x =
        match (switch1 x),(switch2 x) with
        | Success s1,Success s2 -> Success (addSuccess s1 s2)
        | Failure f1,Success _  -> Failure f1
        | Success _ ,Failure f2 -> Failure f2
        | Failure f1,Failure f2 -> Failure (addFailure f1 f2)


/// ===========================================
/// Global types for this project
/// ===========================================
module DomainTypes =

    open CommonLibrary

    /// The DTO for the request
    type Request = {name:string; email:string}

    // Many more types coming soon!

/// ===========================================
/// Logging functions
/// ===========================================
module Logger =

    open CommonLibrary
```

```
    open DomainTypes

    let log twoTrackInput =
        let success x = printfn "DEBUG. Success so far: %A" x; x
        let failure x = printfn "ERROR. %A" x; x
        doubleMap success failure twoTrackInput

/// ===============================================
/// Validation functions
/// ===============================================
module Validation =

    open CommonLibrary
    open DomainTypes

    let validate1 input =
       if input.name = "" then Failure "Name must not be blank"
       else Success input

    let validate2 input =
       if input.name.Length > 50 then Failure "Name must not be longer than 50 chars"
       else Success input

    let validate3 input =
       if input.email = "" then Failure "Email must not be blank"
       else Success input

    // create a "plus" function for validation functions
    let (&&&) v1 v2 =
        let addSuccess r1 r2 = r1 // return first
        let addFailure s1 s2 = s1 + "; " + s2  // concat
        plus addSuccess addFailure v1 v2

    let combinedValidation =
        validate1
        &&& validate2
        &&& validate3

    let canonicalizeEmail input =
       { input with email = input.email.Trim().ToLower() }

/// ===============================================
/// Database functions
/// ===============================================
module CustomerRepository =

    open CommonLibrary
    open DomainTypes

    let updateDatabase input =
       ()   // dummy dead-end function for now

    // new function to handle exceptions
```

```
    let updateDatebaseStep =
        tryCatch (tee updateDatabase) (fun ex -> ex.Message)

/// ===========================================
/// All the use cases or services in one place
/// ===========================================
module UseCases =

    open CommonLibrary
    open DomainTypes

    let handleUpdateRequest =
        Validation.combinedValidation
        >> map Validation.canonicalizeEmail
        >> bind CustomerRepository.updateDatebaseStep
        >> Logger.log
```

Summary

In this post, we looked at organizing code into modules. In the next post in this series, we'll finally start doing some real coding!

Meanwhile, you can read more on cyclic dependencies in the follow up posts:

- Cyclic dependencies are evil.
- Refactoring to remove cyclic dependencies.
- Cycles and modularity in the wild, which compares some real-world metrics for C# and F# projects.

One of the most common complaints about F# is that it requires code to be in *dependency order*. That is, you cannot use forward references to code that hasn't been seen by the compiler yet.

In this series, I discuss dependency cycles, why they are bad, and how to get rid of them.

- Cyclic dependencies are evil. Cyclic dependencies: Part 1.
- Refactoring to remove cyclic dependencies. Cyclic dependencies: Part 2.
- Cycles and modularity in the wild. Comparing some real-world metrics of C# and F# projects.

Cyclic dependencies are evil

One of three related posts on module organization and cyclic dependencies.

One of the most common complaints about F# is that it requires code to be in *dependency order*. That is, you cannot use forward references to code that hasn't been seen by the compiler yet.

Here's a typical example:

> "The order of .fs files makes it hard to compile... My F# application is just over 50 lines of code, but it's already more work than it's worth to compile even the tiniest non-trivial application. Is there a way to make the F# compiler more like the C# compiler, so that it's not so tightly coupled to the order that files are passed to the compiler?" [fpish.net]

and another:

> "After trying to build a slightly above-toy-size project in F#, I came to the conclusion that with current tools it would be quite difficult to maintain a project of even moderate complexity." [www.ikriv.com]

and another:

> "F# compiler [is] too linear. The F# compiler should handle all type resolution matters automatically, independent of declaration order" [www.sturmnet.org]

and one more:

> "The topic of annoying (and IMHO unnecessary) limitations of the F# project system was already discussed on this forum. I am talking about the way compilation order is controlled" [fpish.net]

Well, these complaints are unfounded. You most certainly can build and maintain large projects using F#. The F# compiler and the core library are two obvious examples.

In fact, most of these problems boil down to "why can't F# be like C#". If you are coming from C#, you are used to having the compiler connect everything automatically. Having to deal with dependency relationships explicitly is very annoying -- old-fashioned and regressive, even.

The aim of this post is to explain (a) why dependency management is important, and (b) some techniques that can help you deal with it.

Dependencies are bad things...

We all know that dependencies are the bane of our existence. Assembly dependencies, configuration dependencies, database dependencies, network dependencies -- there's always something.

So we developers, as a profession, tend to put a lot of effort into making dependencies more manageable. This goal manifests itself in many disparate ways: the interface segregation principle, inversion of control and dependency injection; package management with NuGet; configuration management with puppet/chef; and so on. In some sense all these approaches are trying to reduce the number of things we have to be aware of, and the number of things that can break.

This is not a new problem, of course. A large part of the classic book "Large-Scale C++ Software Design" is devoted to dependency management. As John Lakos, the author, put it:

> "The maintenance cost of a subsystem can be reduced significantly by avoiding unnecessary dependencies among components"

The key word here is "unnecessary". What is an "unnecessary" dependency? It depends, of course. But one particular kind of dependency is almost always unnecessary -- a **circular dependency**.

... and circular dependencies are evil

To understand why circular dependencies are evil, let's revisit what we mean by a "component".

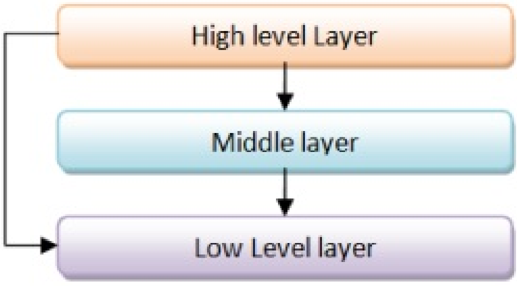
Components are Good Things. Whether you think of them as packages, assemblies, modules, classes or whatever, their primary purpose is to break up large amounts of code into smaller and more manageable pieces. In other words, we are applying a divide and conquer approach to the problem of software development.

But in order to be useful for maintenance, deployment, or whatever, a component shouldn't just be a random collection of stuff. It should (of course) group only *related code* together.

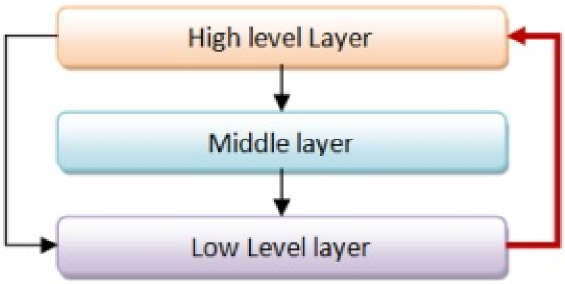
In an ideal world, each component would thus be completely independent of any others. But generally (of course), some dependencies are always necessary.

But, now that we have components with *dependencies*, we need a way to manage these dependencies. One standard way to do this is with the "layering" principle. We can have "high level" layers and "low level" layers, and the critical rule is: *each layer should depend only on layers below it, and never on a layer above it*.

You are very familiar with this, I'm sure. Here's a diagram of some simple layers:



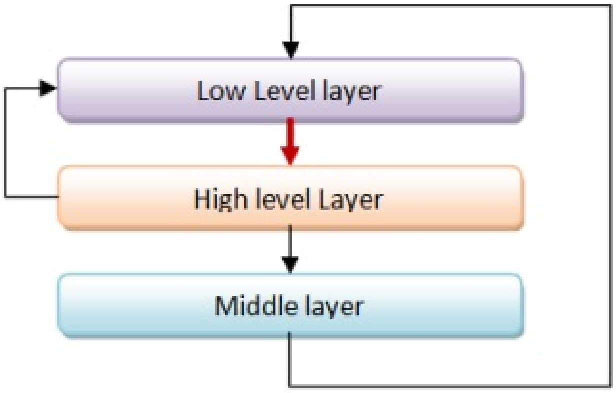
But now what happens when you introduce a dependency from the bottom layer to the top layer, like this?



By having a dependency from the bottom to the top, we have introduced the evil "circular dependency".

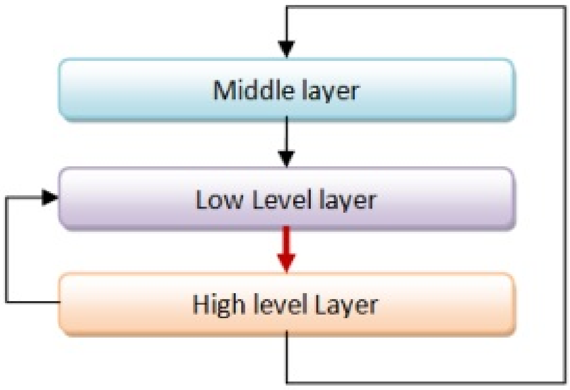
Why is it evil? Because *any* alternative layering method is now valid!

For example, we could put the bottom layer on top instead, like this:



From a logical point of view, this alternative layering is just the same as the original layering.

Or how about we put the middle layer on top?

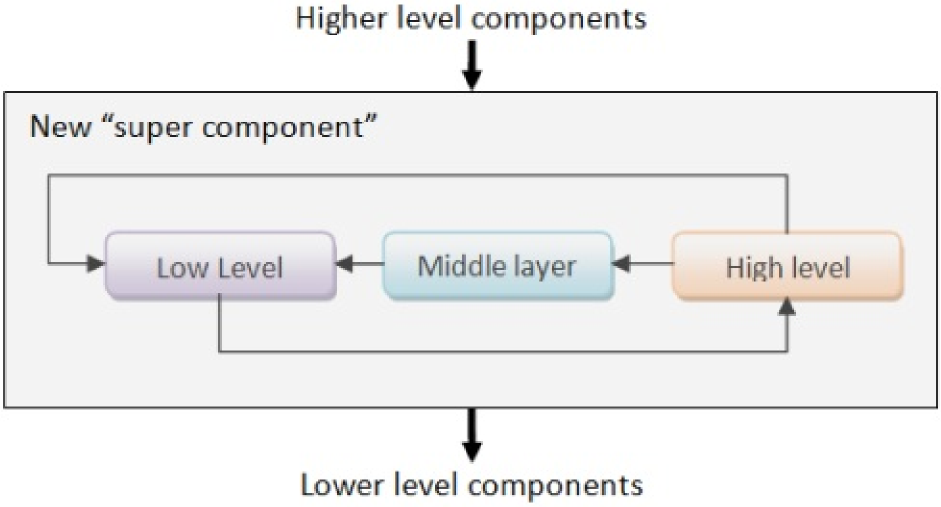


Something has gone badly wrong! It's clear that we've really messed things up.

In fact, as soon as you have any kind of circular dependency between components, the *only* thing you can do is to put them *all* into the *same* layer.



In other words, the circular dependency has completely destroyed our "divide and conquer" approach, the whole reason for having components in the first place. Rather than having three components, we now have just one "super component", which is three times bigger and more complicated than it needed to be.



And that's why circular dependencies are evil.

For more on this subject, see this StackOverflow answer and this article about layering by Patrick Smacchia (of NDepend).

Circular dependencies in the real world

Let's start by looking at circular dependencies between .NET assemblies. Here are some war stories from Brian McNamara (my emphasis):

> The .Net Framework 2.0 has this problem in spades; System.dll, System.Configuration.dll, and System.Xml.dll are all hopelessly entangled with one another. This manifests in a variety of ugly ways. For example, I found a simple [bug] in the VS debugger that effectively crashes the debuggee when hitting a breakpoint while trying to loads symbols, caused by the circular dependencies among these assemblies. Another story: a friend of mine was a developer on the initial versions of Silverlight and was tasked with trying to trim down these three assemblies, and the first arduous task was trying to untangle the circular dependencies. **"Mutual recursion for free" is very convenient on a small scale, but it will destroy you on a large scale.**
>
> VS2008 shipped a week later than planned, because VS2008 had a dependency on SQL server, and SQL server had a dependency on VS, and whoops! in the end they couldn't produce a full product version where everything had the same build number, and had to scramble to make it work. [fpish.net]

So there is plenty of evidence that circular dependencies between assemblies are bad. In fact, circular dependencies between assemblies are considered bad enough that Visual Studio won't even let you create them!

You might say, "Yes, I can understand why circular dependencies are bad for assemblies, but why bother for code inside an assembly?"

Well, for exactly the same reasons! Layering allows better partitioning, easier testing and cleaner refactoring. You can see what I mean in a related post on dependency cycles "in the wild" where I compare C# projects and F# projects. The dependencies in the F# projects are a lot less spaghetti-like.

Another quote from Brian's (excellent) comment:

> I'm evangelizing an unpopular position here, but my experience is that everything in the world is better when you're forced to consider and manage "dependency order among software components" at every level of the system. The specific UI/tooling for F# may not yet be ideal, but I think the principle is right. This is a burden you want. It *is* more work. "Unit testing" is also more work, but we've gotten to the point where the consensus is that work is "worth it" in that it saves you time in the long run. I feel the same way about 'ordering'. There are dependencies among the classes and methods in your system. You ignore those dependencies at your own peril. A system that forces you to consider this dependency graph (roughly, the topological sort of components) is likely to steer you into developing software with cleaner architectures, better system layering, and fewer needless dependencies.

Detecting and removing circular dependencies

Ok, we're agreed that circular dependencies are bad. So how do we detect them and then get rid of them?

Let's start with detection. There are a number of tools to help you detect circular dependencies in your code.

- If you're using C#, you will need a tool like the invaluable NDepend.
- And if you are using Java, there are equivalent tools such as JDepend.
- But if you are using F#, you're in luck! You get circular dependency detection for free!

"Very funny," you might say, "I already know about F#'s circular dependency prohibition -- it's driving me nuts! What can I do to fix the problem and make the compiler happy?"

For that, you'll need to read the next post...

Refactoring to remove cyclic dependencies

In the previous post, we looked at the concept of dependency cycles, and why they are bad.

In this post, we'll look at some techniques for eliminating them from your code. Having to do this may seem annoying at first, but really, you'll come to appreciate that in the long run, "it's not a bug, it's a feature!"

Classifying some common cyclic dependencies

Let's classify the kinds of dependencies you're likely to run into. I'll look at three common situations, and for each one, demonstrate some techniques for dealing with them.

First, there is what I will call a *"method dependency"*.

- Type A stores a value of type B in a property
- Type B references type A in a method signature, but doesn't store a value of type A

Second, there is what I will call a *"structural dependency"*.

- Type A stores a value of type B in a property
- Type B stores a value of type A in a property

Finally, there is what I will call an *"inheritance dependency"*.

- Type A stores a value of type B in a property
- Type B inherits from type A

There are, of course, other variants. But if you know how to deal with these, you can use the same techniques to deal with the others as well.

Three tips on dealing with dependencies in F#

Before we get started, here are three useful tips which apply generally when trying to untangle dependencies.

Tip 1: Treat F# like F#.

Recognize that F# is not C#. If you are willing to work with F# using its native idioms, then it is normally very straightforward to avoid circular dependencies by using a different style of code organization.

Tip 2: Separate types from behavior.

Since most types in F# are immutable, it is acceptable for them to be "exposed" and "anemic", even. So in a functional design it is common to separate the types themselves from the functions that act on them. This approach will often help to clean up dependencies, as we'll see below.

Tip 3: Parameterize, parameterize, parameterize.

Dependencies can only happen when a specific type is referenced. If you use generic types, you cannot have a dependency!

And rather than hard coding behavior for a type, why not parameterize it by passing in functions instead? The `List` module is a great example of this approach, and I'll show some examples below as well.

Dealing with a "method dependency"

We'll start with the simplest kind of dependency -- what I will call a "method dependency".

Here is an example.

```
module MethodDependencyExample =

    type Customer(name, observer:CustomerObserver) =
        let mutable name = name
        member this.Name
            with get() = name
            and set(value) =
                name <- value
                observer.OnNameChanged(this)

    and CustomerObserver() =
        member this.OnNameChanged(c:Customer) =
            printfn "Customer name changed to '%s' " c.Name

    // test
    let observer = new CustomerObserver()
    let customer = Customer("Alice",observer)
    customer.Name <- "Bob"
```

The `Customer` class has a property/field of type `CustomerObserver`, but the `CustomerObserver` class has a method which takes a `Customer` as a parameter, causing a mutual dependency.

Using the "and" keyword

One straightforward way to get the types to compile is to use the `and` keyword, as I did above.

The `and` keyword is designed for just this situation -- it allows you to have two or more types that refer to each other.

To use it, just replace the second `type` keyword with `and`. Note that using `and type`, as shown below, is incorrect. Just the single `and` is all you need.

```
type Something
and type SomethingElse  // wrong

type Something
and SomethingElse       // correct
```

But `and` has a number of problems, and using it is generally discouraged except as a last resort.

First, it only works for types declared in the same module. You can't use it across module boundaries.

Second, it should really only be used for tiny types. If you have 500 lines of code between the `type` and the `and`, then you are doing something very wrong.

```
type Something
   // 500 lines of code
and SomethingElse
   // 500 more lines of code
```

The code snippet shown above is an example of how *not* to do it.

In other words, don't treat `and` as a panacea. Overusing it is a symptom that you have not refactored your code properly.

Introducing parameterization

So, instead of using `and`, let's see what we can do using parameterization, as mentioned in the third tip.

If we think about the example code, do we *really* need a special `CustomerObserver` class? Why have we restricted it to `Customer` only? Can't we have a more generic observer class?

So why don't we create a `INameObserver<'T>` interface instead, with the same `OnNameChanged` method, but the method (and interface) parameterized to accept any class?

Here's what I mean:

```
module MethodDependency_ParameterizedInterface =

    type INameObserver<'T> =
        abstract OnNameChanged : 'T -> unit

    type Customer(name, observer:INameObserver<Customer>) =
        let mutable name = name
        member this.Name
            with get() = name
            and set(value) =
                name <- value
                observer.OnNameChanged(this)

    type CustomerObserver() =
        interface INameObserver<Customer> with
            member this.OnNameChanged c =
                printfn "Customer name changed to '%s' " c.Name

    // test
    let observer = new CustomerObserver()
    let customer = Customer("Alice", observer)
    customer.Name <- "Bob"
```

In this revised version, the dependency has been broken! No `and` is needed at all. In fact, you could even put the types in different projects or assemblies now!

The code is almost identical to the first version, except that the `Customer` constructor accepts a interface, and `CustomerObserver` now implements the same interface. In fact, I would argue that introducing the interface has actually made the code better than before.

But we don't have to stop there. Now that we have an interface, do we really need to create a whole class just to implement it? F# has a great feature called object expressions which allows you to instantiate an interface directly.

Here is the same code again, but this time the `CustomerObserver` class has been eliminated completely and the `INameObserver` created directly.

```
module MethodDependency_ParameterizedInterface =

    // code as above

    // test
    let observer2 = {
        new INameObserver<Customer> with
            member this.OnNameChanged c =
                printfn "Customer name changed to '%s' " c.Name
        }
    let customer2 = Customer("Alice", observer2)
    customer2.Name <- "Bob"
```

This technique will obviously work for more complex interfaces as well, such as that shown below, where there are two methods:

```
module MethodDependency_ParameterizedInterface2 =

    type ICustomerObserver<'T> =
        abstract OnNameChanged : 'T -> unit
        abstract OnEmailChanged : 'T -> unit

    type Customer(name, email, observer:ICustomerObserver<Customer>) =

        let mutable name = name
        let mutable email = email

        member this.Name
            with get() = name
            and set(value) =
                name <- value
                observer.OnNameChanged(this)

        member this.Email
            with get() = email
            and set(value) =
                email <- value
                observer.OnEmailChanged(this)

    // test
    let observer2 = {
        new ICustomerObserver<Customer> with
            member this.OnNameChanged c =
                printfn "Customer name changed to '%s' " c.Name
            member this.OnEmailChanged c =
                printfn "Customer email changed to '%s' " c.Email
        }
    let customer2 = Customer("Alice", "x@example.com",observer2)
    customer2.Name <- "Bob"
    customer2.Email <- "y@example.com"
```

Using functions instead of parameterization

In many cases, we can go even further and eliminate the interface class as well. Why not just pass in a simple function that is called when the name changes, like this:

```
module MethodDependency_ParameterizedClasses_HOF  =

    type Customer(name, observer) =

        let mutable name = name

        member this.Name
            with get() = name
            and set(value) =
                name <- value
                observer this

    // test
    let observer(c:Customer) =
        printfn "Customer name changed to '%s' " c.Name
    let customer = Customer("Alice", observer)
    customer.Name <- "Bob"
```

I think you'll agree that this snippet is "lower ceremony" than either of the previous versions. The observer is now defined inline as needed, very simply:

```
let observer(c:Customer) =
    printfn "Customer name changed to '%s' " c.Name
```

True, it only works when the interface being replaced is simple, but even so, this approach can be used more often than you might think.

A more functional approach: separating types from functions

As I mentioned above, a more "functional design" would be to separate the types themselves from the functions that act on those types. Let's see how this might be done in this case.

Here is a first pass:

```
module MethodDependencyExample_SeparateTypes =

    module DomainTypes =
        type Customer = { name:string; observer:NameChangedObserver }
        and  NameChangedObserver = Customer -> unit


    module Customer =
        open DomainTypes

        let changeName customer newName =
            let newCustomer = {customer with name=newName}
            customer.observer newCustomer
            newCustomer     // return the new customer

    module Observer =
        open DomainTypes

        let printNameChanged customer =
            printfn "Customer name changed to '%s' " customer.name

    // test
    module Test =
        open DomainTypes

        let observer = Observer.printNameChanged
        let customer = {name="Alice"; observer=observer}
        Customer.changeName customer "Bob"
```

In the example above, we now have *three* modules: one for the types, and one each for the functions. Obviously, in a real application, there will be a lot more Customer related functions in the `Customer` module than just this one!

In this code, though, we still have the mutual dependency between `Customer` and `CustomerObserver`. The type definitions are more compact, so it is not such a problem, but even so, can we eliminate the `and`?

Yes, of course. We can use the same trick as in the previous approach, eliminating the observer type and embedding a function directly in the `Customer` data structure, like this:

```
module MethodDependency_SeparateTypes2 =

    module DomainTypes =
        type Customer = { name:string; observer:Customer -> unit}

    module Customer =
        open DomainTypes

        let changeName customer newName =
            let newCustomer = {customer with name=newName}
            customer.observer newCustomer
            newCustomer     // return the new customer

    module Observer =
        open DomainTypes

        let printNameChanged customer =
            printfn "Customer name changed to '%s' " customer.name

    module Test =
        open DomainTypes

        let observer = Observer.printNameChanged
        let customer = {name="Alice"; observer=observer}
        Customer.changeName customer "Bob"
```

Making types dumber

The `Customer` type still has some behavior embedded in it. In many cases, there is no need for this. A more functional approach would be to pass a function only when you need it.

So let's remove the `observer` from the customer type, and pass it as an extra parameter to the `changeName` function, like this:

```
let changeName observer customer newName =
    let newCustomer = {customer with name=newName}
    observer newCustomer    // call the observer with the new customer
    newCustomer             // return the new customer
```

Here's the complete code:

```
module MethodDependency_SeparateTypes3 =

    module DomainTypes =
        type Customer = {name:string}

    module Customer =
        open DomainTypes

        let changeName observer customer newName =
            let newCustomer = {customer with name=newName}
            observer newCustomer    // call the observer with the new customer
            newCustomer             // return the new customer

    module Observer =
        open DomainTypes

        let printNameChanged customer =
            printfn "Customer name changed to '%s' " customer.name

    module Test =
        open DomainTypes

        let observer = Observer.printNameChanged
        let customer = {name="Alice"}
        Customer.changeName observer customer "Bob"
```

You might be thinking that I have made things more complicated now -- I have to specify the `observer` function everywhere I call `changeName` in my code. Surely this is worse than before? At least in the OO version, the observer was part of the customer object and I didn't have to keep passing it in.

Ah, but, you're forgetting the magic of partial application! You can set up a function with the observer "baked in", and then use *that* function everywhere, without needing to pass in an observer every time you use it. Clever!

```
module MethodDependency_SeparateTypes3 =

    // code as above

    module TestWithPartialApplication =
        open DomainTypes

        let observer = Observer.printNameChanged

        // set up this partial application only once (at the top of your module, say)
        let changeName = Customer.changeName observer

        // then call changeName without needing an observer
        let customer = {name="Alice"}
        changeName customer "Bob"
```

But wait... there's more!

Let's look at the `changeName` function again:

```
let changeName observer customer newName =
    let newCustomer = {customer with name=newName}
    observer newCustomer    // call the observer with the new customer
    newCustomer             // return the new customer
```

It has the following steps:

1. do something to make a result value
2. call the observer with the result value
3. return the result value

This is completely generic logic -- it has nothing to do with customers at all. So we can rewrite it as a completely generic library function. Our new function will allow *any* observer function to "hook into" into the result of *any* other function, so let's call it `hook` for now.

```
let hook2 observer f param1 param2 =
    let y = f param1 param2 // do something to make a result value
    observer y              // call the observer with the result value
    y                       // return the result value
```

Actually, I called it `hook2` because the function `f` being "hooked into" has two parameters. I could make another version for functions that have one parameter, like this:

```
let hook observer f param1 =
    let y = f param1 // do something to make a result value
    observer y       // call the observer with the result value
    y                // return the result value
```

If you have read the railway oriented programming post, you might notice that this is quite similar to what I called a "dead-end" function. I won't go into more details here, but this is indeed a common pattern.

Ok, back to the code -- how do we use this generic `hook` function?

- `Customer.changeName` is the function being hooked into, and it has two parameters, so we use `hook2` .
- The observer function is just as before

So, again, we create a partially applied `changeName` function, but this time we create it by passing the observer and the hooked function to `hook2` , like this:

```
let observer = Observer.printNameChanged
let changeName = hook2 observer Customer.changeName
```

Note that the resulting `changeName` has *exactly the same signature* as the original `Customer.changeName` function, so it can be used interchangably with it anywhere.

```
let customer = {name="Alice"}
changeName customer "Bob"
```

Here's the complete code:

```
module MethodDependency_SeparateTypes_WithHookFunction =

    [<AutoOpen>]
    module MyFunctionLibrary =

        let hook observer f param1 =
            let y = f param1 // do something to make a result value
            observer y       // call the observer with the result value
            y                // return the result value

        let hook2 observer f param1 param2 =
            let y = f param1 param2 // do something to make a result value
            observer y              // call the observer with the result value
            y                       // return the result value

    module DomainTypes =
        type Customer = { name:string}

    module Customer =
        open DomainTypes

        let changeName customer newName =
            {customer with name=newName}

    module Observer =
        open DomainTypes

        let printNameChanged customer =
            printfn "Customer name changed to '%s' " customer.name

    module TestWithPartialApplication =
        open DomainTypes

        // set up this partial application only once (at the top of your module, say)
        let observer = Observer.printNameChanged
        let changeName = hook2 observer Customer.changeName

        // then call changeName without needing an observer
        let customer = {name="Alice"}
        changeName customer "Bob"
```

Creating a `hook` function like this might seem to add extra complication initially, but it has eliminated yet more code from the main application, and once you have built up a library of functions like this, you will find uses for them everywhere.

By the way, if it helps you to use OO design terminology, you can think of this approach as a "Decorator" or "Proxy" pattern.

Dealing with a "structural dependency"

The second of our classifications is what I am calling a "structural dependency", where each type stores a value of the other type.

- Type A stores a value of type B in a property
- Type B stores a value of type A in a property

For this set of examples, consider an `Employee` who works at a `Location`. The `Employee` contains the `Location` they work at, and the `Location` stores a list of `Employees` who work there.

Voila -- mutual dependency!

Here is the example in code:

```
module StructuralDependencyExample =

    type Employee(name, location:Location) =
        member this.Name = name
        member this.Location = location

    and Location(name, employees: Employee list) =
        member this.Name = name
        member this.Employees  = employees
```

Before we get on to refactoring, let's consider how awkward this design is. How can we initialize an `Employee` value without having a `Location` value, and vice versa.

Here's one attempt. We create a location with an empty list of employees, and then create other employees using that location:

```
module StructuralDependencyExample =

    // code as above

    module Test =
        let location = new Location("CA",[])
        let alice = new Employee("Alice",location)
        let bob = new Employee("Bob",location)

        location.Employees  // empty!
        |> List.iter (fun employee ->
            printfn "employee %s works at %s" employee.Name employee.Location.Name)
```

But this code doesn't work as we want. We have to set the list of employees for `location` as empty because we can't forward reference the `alice` and `bob` values..

F# will sometimes allow you to use the `and` keyword in these situation too, for recursive "lets". Just as with "type", the "and" keyword replaces the "let" keyword. Unlike "type", the first "let" has to be marked as recursive with `let rec`.

Let's try it. We will give `location` a list of `alice` and `bob` even though they are not declared yet.

```
module UncompilableTest =
    let rec location = new Location("NY",[alice;bob])
    and alice = new Employee("Alice",location  )
    and bob = new Employee("Bob",location )
```

But no, the compiler is not happy about the infinite recursion that we have created. In some cases, `and` does indeed work for `let` definitions, but this is not one of them! And anyway, just as for types, having to use `and` for "let" definitions is a clue that you might need to refactor.

So, really, the only sensible solution is to use mutable structures, and to fix up the location object *after* the individual employees have been created, like this:

```
module StructuralDependencyExample_Mutable =

    type Employee(name, location:Location) =
        member this.Name = name
        member this.Location = location

    and Location(name, employees: Employee list) =
        let mutable employees = employees

        member this.Name = name
        member this.Employees  = employees
        member this.SetEmployees es =
            employees <- es

    module TestWithMutableData =
        let location = new Location("CA",[])
        let alice = new Employee("Alice",location)
        let bob = new Employee("Bob",location)
        // fixup after creation
        location.SetEmployees [alice;bob]

        location.Employees
        |> List.iter (fun employee ->
            printfn "employee %s works at %s" employee.Name employee.Location.Name)
```

So, a lot of trouble just to create some values. This is another reason why mutual dependencies are a bad idea!

Parameterizing again

To break the dependency, we can use the parameterization trick again. We can just create a parameterized vesion of `Employee`.

```
module StructuralDependencyExample_ParameterizedClasses =

    type ParameterizedEmployee<'Location>(name, location:'Location) =
        member this.Name = name
        member this.Location = location

    type Location(name, employees: ParameterizedEmployee<Location> list) =
        let mutable employees = employees
        member this.Name = name
        member this.Employees  = employees
        member this.SetEmployees es =
            employees <- es

    type Employee = ParameterizedEmployee<Location>

    module Test =
        let location = new Location("CA",[])
        let alice = new Employee("Alice",location)
        let bob = new Employee("Bob",location)
        location.SetEmployees [alice;bob]

        location.Employees  // non-empty!
        |> List.iter (fun employee ->
            printfn "employee %s works at %s" employee.Name employee.Location.Name)
```

Note that we create a type alias for `Employee`, like this:

```
type Employee = ParameterizedEmployee<Location>
```

One nice thing about creating an alias like that is that the original code for creating employees will continue to work unchanged.

```
let alice = new Employee("Alice",location)
```

Parameterizing with behavior dependencies

The code above assumes that the particular class being parameterized over is not important. But what if there are dependencies on particular properties of the type?

For example, let's say that the `Employee` class expects a `Name` property, and the `Location` class expects an `Age` property, like this:

```
module StructuralDependency_WithAge =

    type Employee(name, age:float, location:Location) =
        member this.Name = name
        member this.Age = age
        member this.Location = location

        // expects Name property
        member this.LocationName = location.Name

    and Location(name, employees: Employee list) =
        let mutable employees = employees
        member this.Name = name
        member this.Employees  = employees
        member this.SetEmployees es =
            employees <- es

        // expects Age property
        member this.AverageAge =
            employees |> List.averageBy (fun e -> e.Age)

    module Test =
        let location = new Location("CA",[])
        let alice = new Employee("Alice",20.0,location)
        let bob = new Employee("Bob",30.0,location)
        location.SetEmployees [alice;bob]
        printfn "Average age is %g" location.AverageAge
```

How can we possibly parameterize this?

Well, let's try using the same approach as before:

```
module StructuralDependencyWithAge_ParameterizedError =

    type ParameterizedEmployee<'Location>(name, age:float, location:'Location) =
        member this.Name = name
        member this.Age = age
        member this.Location = location
        member this.LocationName = location.Name  // error

    type Location(name, employees: ParameterizedEmployee<Location> list) =
        let mutable employees = employees
        member this.Name = name
        member this.Employees  = employees
        member this.SetEmployees es =
            employees <- es
        member this.AverageAge =
            employees |> List.averageBy (fun e -> e.Age)
```

The `Location` is happy with `ParameterizedEmployee.Age`, but `location.Name` fails to compile. obviously, because the type parameter is too generic.

One way would be to fix this by creating interfaces such as `ILocation` and `IEmployee`, and that might often be the most sensible approach.

But another way is to let the Location parameter be generic and pass in an *additional function* that knows how to handle it. In this case a `getLocationName` function.

```
module StructuralDependencyWithAge_ParameterizedCorrect =

    type ParameterizedEmployee<'Location>(name, age:float, location:'Location, getLoca
tionName) =
        member this.Name = name
        member this.Age = age
        member this.Location = location
        member this.LocationName = getLocationName location  // ok

    type Location(name, employees: ParameterizedEmployee<Location> list) =
        let mutable employees = employees
        member this.Name = name
        member this.Employees  = employees
        member this.SetEmployees es =
            employees <- es
        member this.AverageAge =
            employees |> List.averageBy (fun e -> e.Age)
```

One way of thinking about this is that we are providing the behavior externally, rather than as part of the type.

To use this then, we need to pass in a function along with the type parameter. This would be annoying to do all the time, so naturally we will wrap it in a function, like this:

```
module StructuralDependencyWithAge_ParameterizedCorrect =

    // same code as above

    // create a helper function to construct Employees
    let Employee(name, age, location) =
        let getLocationName (l:Location) = l.Name
        new ParameterizedEmployee<Location>(name, age, location, getLocationName)
```

With this in place, the original test code continues to work, almost unchanged (we have to change `new Employee` to just `Employee`).

```
module StructuralDependencyWithAge_ParameterizedCorrect =

    // same code as above

    module Test =
        let location = new Location("CA",[])
        let alice = Employee("Alice",20.0,location)
        let bob = Employee("Bob",30.0,location)
        location.SetEmployees [alice;bob]

        location.Employees  // non-empty!
        |> List.iter (fun employee ->
            printfn "employee %s works at %s" employee.Name employee.LocationName)
```

The functional approach: separating types from functions again

Now let's apply the functional design approach to this problem, just as we did before.

Again, we'll separate the types themselves from the functions that act on those types.

```
module StructuralDependencyExample_SeparateTypes =

    module DomainTypes =
        type Employee = {name:string; age:float; location:Location}
        and Location = {name:string; mutable employees: Employee list}

    module Employee =
        open DomainTypes

        let Name (employee:Employee) = employee.name
        let Age (employee:Employee) = employee.age
        let Location (employee:Employee) = employee.location
        let LocationName (employee:Employee) = employee.location.name

    module Location =
        open DomainTypes

        let Name (location:Location) = location.name
        let Employees (location:Location) = location.employees
        let AverageAge (location:Location) =
            location.employees |> List.averageBy (fun e -> e.age)

    module Test =
        open DomainTypes

        let location = { name="NY"; employees= [] }
        let alice = {name="Alice"; age=20.0; location=location  }
        let bob = {name="Bob"; age=30.0; location=location }
        location.employees <- [alice;bob]

        Location.Employees location
        |> List.iter (fun e ->
            printfn "employee %s works at %s" (Employee.Name e) (Employee.LocationName
 e) )
```

Before we go any further, let's remove some unneeded code. One nice thing about using a record type is that you don't need to define "getters", so the only functions you need in the modules are functions that manipulate the data, such as `AverageAge`.

```
module StructuralDependencyExample_SeparateTypes2 =

    module DomainTypes =
        type Employee = {name:string; age:float; location:Location}
        and Location = {name:string; mutable employees: Employee list}

    module Employee =
        open DomainTypes

        let LocationName employee = employee.location.name

    module Location =
        open DomainTypes

        let AverageAge location =
            location.employees |> List.averageBy (fun e -> e.age)
```

Parameterizing again

Once again, we can remove the dependency by creating a parameterized version of the types.

Let's step back and think about the "location" concept. Why does a location have to only contain Employees? If we make it a bit more generic, we could consider a location as being a "place" plus "a list of things at that place".

For example, if the things are products, then a place full of products might be a warehouse. If the things are books, then a place full of books might be a library.

Here are these concepts expressed in code:

```
module LocationOfThings =

    type Location<'Thing> = {name:string; mutable things: 'Thing list}

    type Employee = {name:string; age:float; location:Location<Employee> }
    type WorkLocation = Location<Employee>

    type Product = {SKU:string; price:float }
    type Warehouse = Location<Product>

    type Book = {title:string; author:string}
    type Library = Location<Book>
```

Of course, these locations are not exactly the same, but there might be something in common that you can extract into a generic design, especially as there is no behavior requirement attached to the things they contain.

So, using the "location of things" design, here is our dependency rewritten to use parameterized types.

```
module StructuralDependencyExample_SeparateTypes_Parameterized =

    module DomainTypes =
        type Location<'Thing> = {name:string; mutable things: 'Thing list}
        type Employee = {name:string; age:float; location:Location<Employee> }

    module Employee =
        open DomainTypes

        let LocationName employee = employee.location.name

    module Test =
        open DomainTypes

        let location = { name="NY"; things = [] }
        let alice = {name="Alice"; age=20.0; location=location  }
        let bob = {name="Bob"; age=30.0; location=location }
        location.things <- [alice;bob]

        let employees = location.things
        employees
        |> List.iter (fun e ->
            printfn "employee %s works at %s" (e.name) (Employee.LocationName e) )

        let averageAge =
            employees
            |> List.averageBy (fun e -> e.age)
```

In this revised design you will see that the `AverageAge` function has been completely removed from the `Location` module. There is really no need for it, because we can do these kinds of calculations quite well "inline" without needing the overhead of special functions.

And if you think about it, if we *did* need to have such a function pre-defined, it would probably be more appropriate to put in the `Employee` module rather than the `Location` module. After all, the functionality is much more related to how employees work than how locations work.

Here's what I mean:

```
module Employee =

    let AverageAgeAtLocation location =
        location.things |> List.averageBy (fun e -> e.age)
```

This is one advantage of modules over classes; you can mix and match functions with different types, as long as they are all related to the underlying use cases.

Moving relationships into distinct types

In the examples so far, the "list of things" field in location has had to be mutable. How can we work with immutable types and still support relationships?

Well one way *not* to do it is to have the kind of mutual dependency we have seen. In that design, synchronization (or lack of) is a terrible problem

For example, I could change Alice's location without telling the location she points to, resulting in an inconsistency. But if I tried to change the contents of the location as well, then I would also need to update the value of Bob as well. And so on, ad infinitum. A nightmare, basically.

The correct way to do this with immutable data is steal a leaf from database design, and extract the relationship into a separate "table" or type in our case. The current relationships are held in a single master list, and so when changes are made, no synchronization is needed.

Here is a very crude example, using a simple list of `Relationship`s.

```
module StructuralDependencyExample_Normalized =

    module DomainTypes =
        type Relationship<'Left,'Right> = 'Left * 'Right

        type Location= {name:string}
        type Employee = {name:string; age:float }

    module Employee =
        open DomainTypes

        let EmployeesAtLocation location relations =
            relations
            |> List.filter (fun (loc,empl) -> loc = location)
            |> List.map (fun (loc,empl) -> empl)

        let AverageAgeAtLocation location relations =
            EmployeesAtLocation location relations
            |> List.averageBy (fun e -> e.age)

    module Test =
        open DomainTypes

        let location = { Location.name="NY"}
        let alice = {name="Alice"; age=20.0; }
        let bob = {name="Bob"; age=30.0; }
        let relations = [
            (location,alice)
            (location,bob)
            ]

        relations
        |> List.iter (fun (loc,empl) ->
            printfn "employee %s works at %s" (empl.name) (loc.name) )
```

Or course, a more efficient design would use dictionaries/maps, or special in-memory structures designed for this kind of thing.

Inheritance dependencies

Finally, let's look at an "inheritance dependency".

- Type A stores a value of type B in a property
- Type B inherits from type A

We'll consider a UI control hierarchy, where every control belongs to a top-level "Form", and the Form itself is a Control.

Here's a first pass at an implementation:

```
module InheritanceDependencyExample =

    type Control(name, form:Form) =
        member this.Name = name

        abstract Form : Form
        default this.Form = form

    and Form(name) as self =
        inherit Control(name, self)

    // test
    let form = new Form("form")       // NullReferenceException!
    let button = new Control("button",form)
```

The thing to note here is that the Form passes itself in as the `form` value for the Control constructor.

This code will compile, but will cause a `NullReferenceException` error at runtime. This kind of technique will work in C#, but not in F#, because the class initialization logic is done differently.

Anyway, this is a terrible design. The form shouldn't have to pass itself in to a constructor.

A better design, which also fixes the constructor error, is to make `Control` an abstract class instead, and distinguish between non-form child classes (which do take a form in their constructor) and the `Form` class itself, which doesn't.

Here's some sample code:

```
module InheritanceDependencyExample2 =

    [<AbstractClass>]
    type Control(name) =
        member this.Name = name

        abstract Form : Form

    and Form(name) =
        inherit Control(name)

        override this.Form = this

    and Button(name,form) =
        inherit Control(name)

        override this.Form = form

    // test
    let form = new Form("form")
    let button = new Button("button",form)
```

Our old friend parameterization again

To remove the circular dependency, we can parameterize the classes in the usual way, as shown below.

```
module InheritanceDependencyExample_ParameterizedClasses =

    [<AbstractClass>]
    type Control<'Form>(name) =
        member this.Name = name

        abstract Form : 'Form

    type Form(name) =
        inherit Control<Form>(name)

        override this.Form = this

    type Button(name,form) =
        inherit Control<Form>(name)

        override this.Form = form


    // test
    let form = new Form("form")
    let button = new Button("button",form)
```

A functional version

I will leave a functional design as an exercise for you to do yourself.

If we were going for truly functional design, we probably would not be using inheritance at all. Instead, we would use composition in conjunction with parameterization.

But that's a big topic, so I'll save it for another day.

Summary

I hope that this post has given you some useful tips on removing dependency cycles. With these various approaches in hand, any problems with module organization should be able to be resolved easily.

In the next post in this series, I'll look at dependency cycles "in the wild", by comparing some real world C# and F# projects.

As we have seen, F# is a very opinionated language! It wants us to use modules instead of classes and it prohibits dependency cycles. Are these just annoyances, or do they really make a difference to the way that code is organized? Read on and find out!

Cycles and modularity in the wild

(*Updated 2013-06-15. See comments at the end of the post*)

(*Updated 2014-04-12. A follow up post that applies the same analysis to Roslyn*)

(*Updated 2015-01-23. A much clearer version of this analysis has been done by Evelina Gabasova. She knows what she is talking about, so I highly recommend you read her post first!*)

This is a follow up post to two earlier posts on module organization and cyclic dependencies.

I thought it would be interesting to look at some real projects written in C# and F#, and see how they compare in modularity and number of cyclic dependencies.

The plan

My plan was to take ten or so projects written in C# and ten or so projects written in F#, and somehow compare them.

I didn't want to spend too much time on this, and so rather than trying to analyze the source files, I thought I would cheat a little and analyze the compiled assemblies, using the Mono.Cecil library.

This also meant that I could get the binaries directly, using NuGet.

The projects I picked were:

C# projects

- Mono.Cecil, which inspects programs and libraries in the ECMA CIL format.
- NUnit
- SignalR for real-time web functionality.
- NancyFx, a web framework
- YamlDotNet, for parsing and emitting YAML.
- SpecFlow, a BDD tool.
- Json.NET.
- Entity Framework.
- ELMAH, a logging framework for ASP.NET.
- NuGet itself.
- Moq, a mocking framework.
- NDepend, a code analysis tool.

- And, to show I'm being fair, a business application that I wrote in C#.

F# projects

Unfortunately, there is not yet a wide variety of F# projects to choose from. I picked the following:

- FSharp.Core, the core F# library.
- FSPowerPack.
- FsUnit, extensions for NUnit.
- Canopy, a wrapper around the Selenium test automation tool.
- FsSql, a nice little ADO.NET wrapper.
- WebSharper, the web framework.
- TickSpec, a BDD tool.
- FSharpx, an F# library.
- FParsec, a parser library.
- FsYaml, a YAML library built on FParsec.
- Storm, a tool for testing web services.
- Foq, a mocking framework.
- Another business application that I wrote, this time in F#.

I did choose SpecFlow and TickSpec as being directly comparable, and also Moq and and Foq.

But as you can see, most of the F# projects are not directly comparable to the C# ones. For example, there is no direct F# equivalent to Nancy, or Entity Framework.

Nevertheless, I was hoping that I might observe some sort of pattern by comparing the projects. And I was right. Read on for the results!

What metrics to use?

I wanted to examine two things: "modularity" and "cyclic dependencies".

First, what should be the unit of "modularity"?

From a coding point of view, we generally work with files (Smalltalk being a notable exception), and so it makes sense to think of the *file* as the unit of modularity. A file is used to group related items together, and if two chunks of code are in different files, they are somehow not as "related" as if they were in the same file.

In C#, the best practice is to have one class per file. So 20 files means 20 classes. Sometimes classes have nested classes, but with rare exceptions, the nested class is in the same file as the parent class. This means that we can ignore them and just use top-level

classes as our unit of modularity, as a proxy for files.

In F#, the best practice is to have one *module* per file (or sometimes more). So 20 files means 20 modules. Behind the scenes, modules are turned into static classes, and any classes defined within the module are turned into nested classes. So again, this means that we can ignore nested classes and just use top-level classes as our unit of modularity.

The C# and F# compilers generate many "hidden" types, for things such as LINQ, lambdas, etc. In some cases, I wanted to exclude these, and only include "authored" types, which have been coded for explicitly. I also excluded the case classes generated by F# discriminated unions from being "authored" classes as well. That means that a union type with three cases will be counted as one authored type rather than four.

So my definition of a *top-level type* is: a type that is not nested and which is not compiler generated.

The metrics I chose for modularity were:

- **The number of top-level types** as defined above.
- **The number of authored types** as defined above.
- **The number of all types**. This number would include the compiler generated types as well. Comparing this number to the top-level types gives us some idea of how representative the top-level types are.
- **The size of the project**. Obviously, there will be more types in a larger project, so we need to make adjustments based on the size of the project. The size metric I picked was the number of instructions, rather than the physical size of the file. This eliminates issues with embedded resources, etc.

Dependencies

Once we have our units of modularity, we can look at dependencies between modules.

For this analysis, I only want to include dependencies between types in the same assembly. In other words, dependencies on system types such as `String` or `List` do not count as a dependency.

Let's say we have a top-level type `A` and another top-level type `B`. Then I say that a *dependency* exists from `A` to `B` if:

- Type `A` or any of its nested types inherits from (or implements) type `B` or any of its nested types.
- Type `A` or any of its nested types has a field, property or method that references type `B` or any of its nested types as a parameter or return value. This includes private members as well -- after all, it is still a dependency.

- Type `A` or any of its nested types has a method implementation that references type `B` or any of its nested types.

This might not be a perfect definition. But it is good enough for my purposes.

In addition to all dependencies, I thought it might be useful to look at "public" or "published" dependencies. A *public dependency* from `A` to `B` exists if:

- Type `A` or any of its nested types inherits from (or implements) type `B` or any of its nested types.
- Type `A` or any of its nested types has a *public* field, property or method that references type `B` or any of its nested types as a parameter or return value.
- Finally, a public dependency is only counted if the source type itself is public.

The metrics I chose for dependencies were:

- **The total number of dependencies**. This is simply the sum of all dependencies of all types. Again, there will be more dependencies in a larger project, but we will also take the size of the project into account.
- **The number of types that have more than X dependencies**. This gives us an idea of how many types are "too" complex.

Cyclic dependencies

Given this definition of dependency, then, a *cyclic dependency* occurs when two different top-level types depend on each other.

Note what *not* included in this definition. If a nested type in a module depends on another nested type in the *same* module, then that is not a cyclic dependency.

If there is a cyclic dependency, then there is a set of modules that are all linked together. For example, if `A` depends on `B`, `B` depends on `C`, and then say, `C` depends on `A`, then `A`, `B` and `C` are linked together. In graph theory, this is called a *strongly connected component*.

The metrics I chose for cyclic dependencies were:

- **The number of cycles**. That is, the number of strongly connected components which had more than one module in them.
- **The size of the largest component.** This gives us an idea of how complex the dependencies are.

I analyzed cyclic dependencies for all dependencies and also for public dependencies only.

Doing the experiment

First, I downloaded each of the project binaries using NuGet. Then I wrote a little F# script that did the following steps for each assembly:

1. Analyzed the assembly using Mono.Cecil and extracted all the types, including the nested types
2. For each type, extracted the public and implementation references to other types, divided into internal (same assembly) and external (different assembly).
3. Created a list of the "top level" types.
4. Created a dependency list from each top level type to other top level types, based on the lower level dependencies.

This dependency list was then used to extract various statistics, shown below. I also rendered the dependency graphs to SVG format (using graphViz).

For cycle detection, I used the QuickGraph library to extract the strongly connected components, and then did some more processing and rendering.

If you want the gory details, here is a link to the script that I used, and here is the raw data.

It is important to recognize that this is *not* a proper statistical study, just a quick analysis. However the results are quite interesting, as we shall see.

Modularity

Let's look at the modularity first.

Here are the modularity-related results for the C# projects:

Project	Code size	Top-level types	Authored types	All types	Code/Top	Code/Auth
ef	269521	514	565	876	524	477
jsonDotNet	148829	215	232	283	692	642
nancy	143445	339	366	560	423	392
cecil	101121	240	245	247	421	413
nuget	114856	216	237	381	532	485
signalR	65513	192	229	311	341	286
nunit	45023	173	195	197	260	231
specFlow	46065	242	287	331	190	161
elmah	43855	116	140	141	378	313
yamlDotNet	23499	70	73	73	336	322
fparsecCS	57474	41	92	93	1402	625
moq	133189	397	420	533	335	317
ndepend	478508	734	828	843	652	578
ndependPlat	151625	185	205	205	820	740
personalCS	422147	195	278	346	2165	1519
TOTAL	2244670	3869	4392	5420	580	511

And here are the results for the F# projects:

Project	Code size	Top-level types	Authored types	All types	Code/Top	Code/Autl
fsxCore	339596	173	328	2024	1963	1035
fsCore	226830	154	313	1186	1473	725
fsPowerPack	117581	93	150	410	1264	784
storm	73595	67	70	405	1098	1051
fParsec	67252	8	24	245	8407	2802
websharper	47391	52	128	285	911	370
tickSpec	30797	34	49	170	906	629
websharperHtml	14787	18	28	72	822	528
canopy	15105	6	16	103	2518	944
fsYaml	15191	7	11	160	2170	1381
fsSql	15434	13	18	162	1187	857
fsUnit	1848	2	3	7	924	616
foq	26957	35	48	103	770	562
personalFS	118893	30	146	655	3963	814
TOTAL	1111257	692	1332	5987	1606	834

The columns are:

- **Code size** is the number of CIL instructions from all methods, as reported by Cecil.
- **Top-level types** is the total number of top-level types in the assembly, using the definition above.
- **Authored types** is the total number of types in the assembly, including nested types, enums, and so on, but excluding compiler generated types.
- **All types** is the total number of types in the assembly, including compiler generated types.

I have extended these core metrics with some extra calculated columns:

- **Code/Top** is the number of CIL instructions per top level type / module. This is a measure of how much code is associated with each unit of modularity. Generally, more is better, because you don't want to have to deal with multiple files if you don't have too. On the other hand, there is a trade off. Too many lines of code in a file makes reading the code impossible. In both C# and F#, good practice is not to have more than 500-1000 lines of code per file, and with a few exceptions, that seems to be the case in the source code that I looked at.

- **Code/Auth** is the number of CIL instructions per authored type. This is a measure of how "big" each authored type is.
- **Code/All** is the number of CIL instructions per type. This is a measure of how "big" each type is.
- **Auth/Top** is the ratio of all authored types to the top-level-types. It is a rough measure of how many authored types are in each unit of modularity.
- **All/Top** is the ratio of all types to the top-level-types. It is a rough measure of how many types are in each unit of modularity.

Analysis

The first thing I noticed is that, with a few exceptions, the code size is bigger for the C# projects than for the F# projects. Partly that is because I picked bigger projects, of course. But even for a somewhat comparable project like SpecFlow vs. TickSpec, the SpecFlow code size is bigger. It may well be that SpecFlow does a lot more than TickSpec, of course, but it also may be a result of using more generic code in F#. There is not enough information to know either way right now -- it would be interesting to do a true side by side comparison.

Next, the number of top-level types. I said earlier that this should correspond to the number of files in a project. Does it?

I didn't get all the sources for all the projects to do a thorough check, but I did a couple of spot checks. For example, for Nancy, there are 339 top level classes, which implies that there should be about 339 files. In fact, there are actually 322 .cs files, so not a bad estimate.

On the other hand, for SpecFlow there are 242 top level types, but only 171 .cs files, so a bit of an overestimate there. And for Cecil, the same thing: 240 top level classes but only 128 .cs files.

For the FSharpX project, there are 173 top level classes, which implies there should be about 173 files. In fact, there are actually only 78 .fs files, so it is a serious over-estimate by a factor of more than 2. And if we look at Storm, there are 67 top level classes. In fact, there are actually only 35 .fs files, so again it is an over-estimate by a factor of 2.

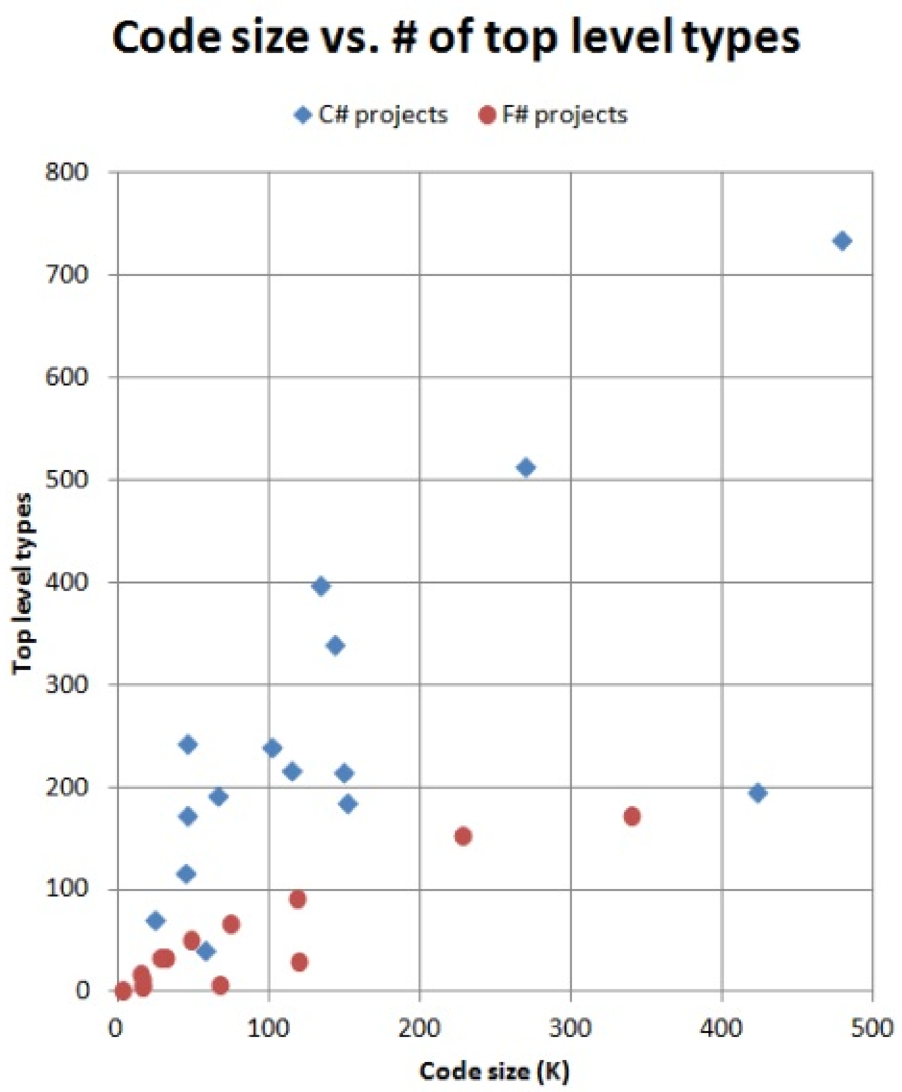
So it looks like the number of top level classes is always an over-estimate of the number of files, but much more so for F# than for C#. It would be worth doing some more detailed analysis in this area.

Ratio of code size to number of top-level types

The "Code/Top" ratio is consistently bigger for F# code than for C# code. Overall, the average top-level type in C# is converted into 580 instructions. But for F# that number is 1606 instructions, about three times as many.

I expect that this is because F# code is more concise than C# code. I would guess that 500 lines of F# code in a single module would create many more CIL instructions than 500 lines of C# code in a class.

If we visually plot "Code size" vs. "Top-level types", we get this chart:



What's surprising to me is how distinct the F# and C# projects are in this chart. The C# projects seem to have a consistent ratio of about 1-2 top-level types per 1000 instructions, even across different project sizes. And the F# projects are consistent too, having a ratio of about 0.6 top-level types per 1000 instructions.

In fact, the number of top level types in F# projects seems to taper off as projects get larger, rather than increasing linearly like the C# projects.

The message I get from this chart is that, for a given size of project, an F# implementation will have fewer modules, and presumably less complexity as a result.

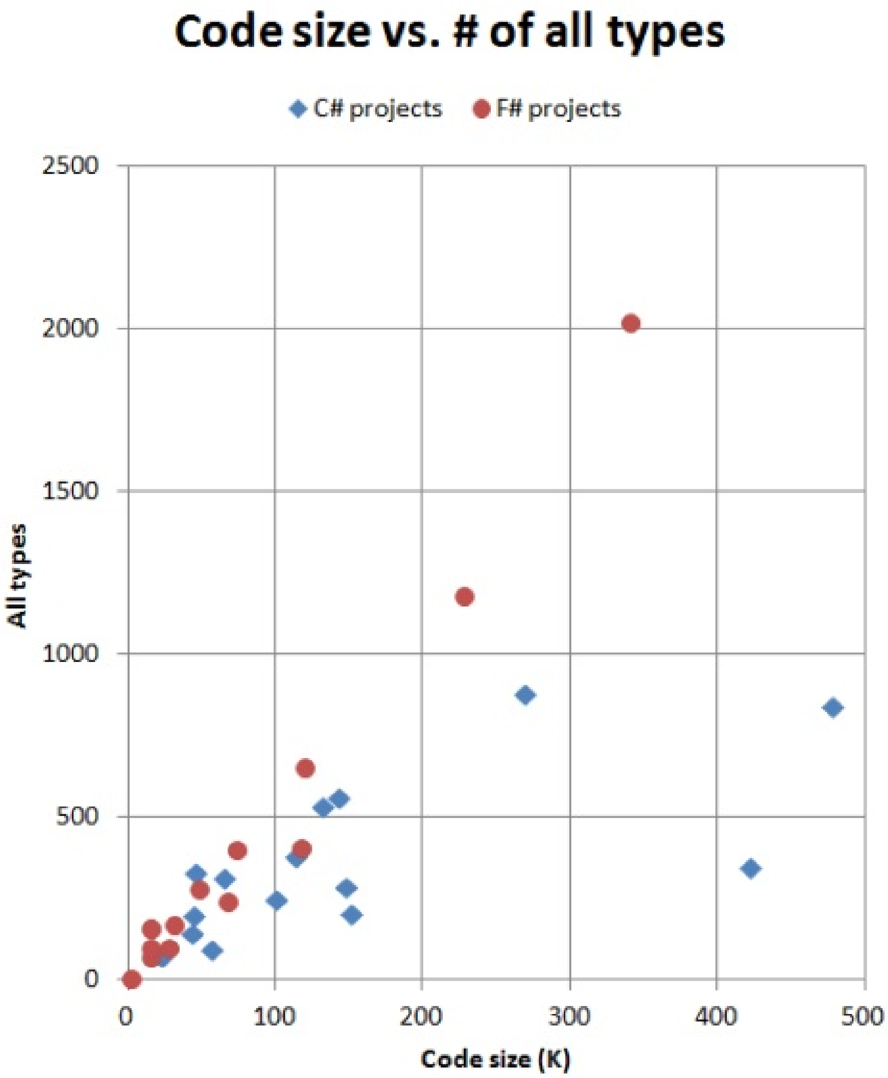
You probably noticed that there are two anomalies. Two C# projects are out of place -- the one at the 50K mark is FParsecCS and the one at the 425K mark is my business application.

I am fairly certain that this because both these implementations have some rather large C# classes in them, which helps the code ratio. Probably a necessarily evil for a parser, but in the case of my business application, I know that it is due to cruft accumulating over the years, and there are some massive classes that ought to be refactored into smaller ones. So a metric like this is probably a *bad* sign for a C# code base.

Ratio of code size to number of all types

On the other hand, if we compare the ratio of code to all types, including compiler generated ones, we get a very different result.

Here's the corresponding chart of "Code size" vs. "All types":



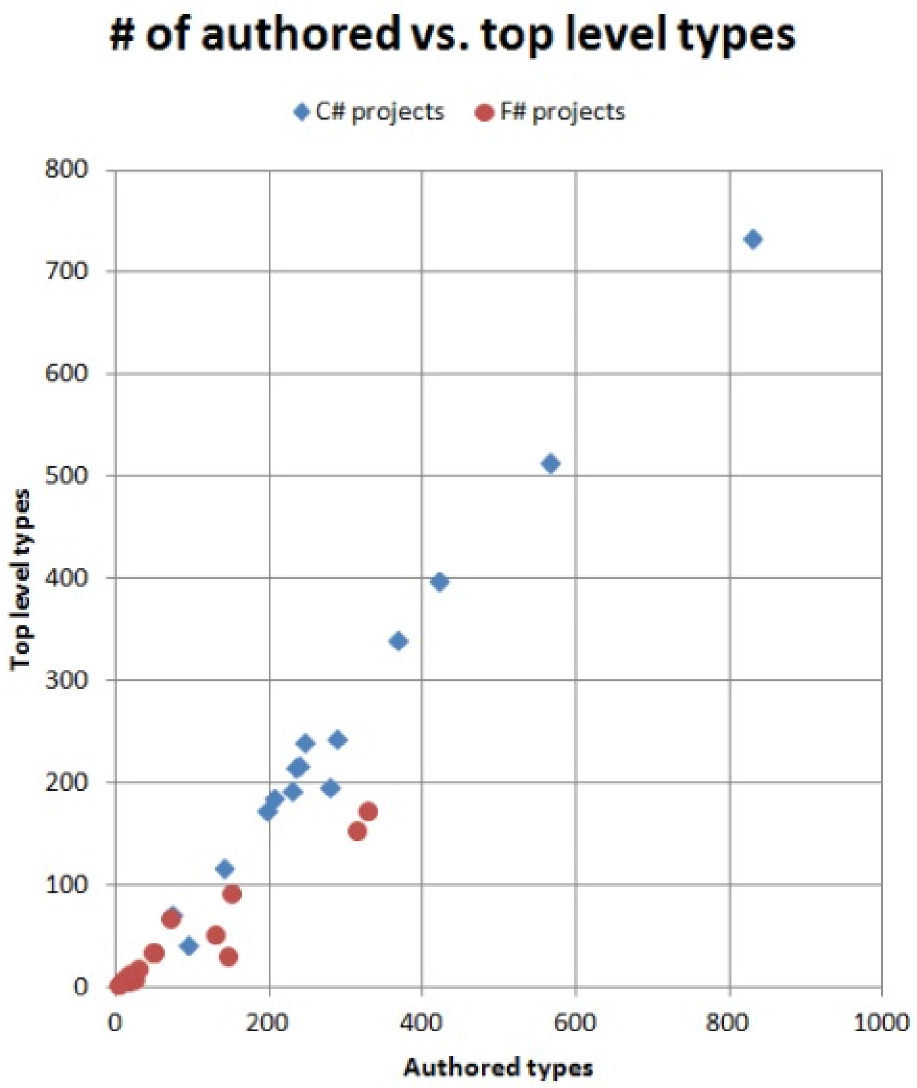
This is surprisingly linear for F#. The total number of types (including compiler generated ones) seems to depend closely on the size of the project. On the other hand, the number of types for C# seems to vary a lot.

The average "size" of a type is somewhat smaller for F# code than for C# code. The average type in C# is converted into about 400 instructions. But for F# that number is about 180 instructions.

I'm not sure why this is. Is it because the F# types are more fine-grained, or could it be because the F# compiler generates many more little types than the C# compiler? Without doing a more subtle analysis, I can't tell.

Ratio of top-level types to authored types

Having compared the type counts to the code size, let's now compare them to each other:



Again, there is a significant difference. For each unit of modularity in C# there are an average of 1.1 authored types. But in F# the average is 1.9, and for some projects a lot more than that.

Of course, creating nested types is trivial in F#, and quite uncommon in C#, so you could argue that this is not a fair comparison. But surely the ability to create a dozen types in as many lines of F# has some effect on the quality of the design? This is harder to do in C#, but there is nothing to stop you. So might this not mean that there is a temptation in C# to not be as fine-grained as you could potentially be?

The project with the highest ratio (4.9) is my F# business application. I believe that this is due to this being only F# project in this list which is designed around a specific business domain, I created many "little" types to model the domain accurately, using the concepts described here. For other projects created using DDD principles, I would expect to see this same high number.

Dependencies

Now let's look at the dependency relationships between the top level classes.

Here are the results for the C# projects:

Project	Top Level Types	Total Dep. Count	Dep/Top	One or more dep.	Three or more dep.	Five or more dep.	Ten or more dep.	D
ef	514	2354	4.6	76%	51%	32%	13%	sv
jsonDotNet	215	913	4.2	69%	42%	30%	14%	sv
nancy	339	1132	3.3	78%	41%	22%	6%	sv
cecil	240	1145	4.8	73%	43%	23%	13%	sv
nuget	216	833	3.9	71%	43%	26%	12%	sv
signalR	192	641	3.3	66%	34%	19%	10%	sv
nunit	173	499	2.9	75%	39%	13%	4%	sv
specFlow	242	578	2.4	64%	25%	17%	5%	sv
elmah	116	300	2.6	72%	28%	22%	6%	sv
yamlDotNet	70	228	3.3	83%	30%	11%	4%	sv
fparsecCS	41	64	1.6	59%	29%	5%	0%	sv
moq	397	1100	2.8	63%	29%	17%	7%	sv
ndepend	734	2426	3.3	67%	37%	25%	10%	sv
ndependPlat	185	404	2.2	67%	24%	11%	4%	sv
personalCS	195	532	2.7	69%	29%	19%	7%	
TOTAL	3869	13149	3.4	70%	37%	22%	9%	

And here are the results for the F# projects:

Project	Top Level Types	Total Dep. Count	Dep/Top	One or more dep.	Three or more dep.	Five or more dep.	Ten or more dep.
fsxCore	173	76	0.4	30%	4%	1%	0%
fsCore	154	287	1.9	55%	26%	14%	3%
fsPowerPack	93	68	0.7	38%	13%	2%	0%
storm	67	195	2.9	72%	40%	18%	4%
fParsec	8	9	1.1	63%	25%	0%	0%
websharper	52	18	0.3	31%	0%	0%	0%
tickSpec	34	48	1.4	50%	15%	9%	3%
websharperHtml	18	37	2.1	78%	39%	6%	0%
canopy	6	8	1.3	50%	33%	0%	0%
fsYaml	7	10	1.4	71%	14%	0%	0%
fsSql	13	14	1.1	54%	8%	8%	0%
fsUnit	2	0	0.0	0%	0%	0%	0%
foq	35	66	1.9	66%	29%	11%	0%
personalFS	30	111	3.7	93%	60%	27%	7%
TOTAL	692	947	1.4	49%	19%	8%	1%

The columns are:

- **Top-level types** is the total number of top-level types in the assembly, as before.
- **Total dep. count** is the total number of dependencies between top level types.
- **Dep/Top** is the number of dependencies per top level type / module only. This is a measure of how many dependencies the average top level type/module has.
- **One or more dep** is the number of top level types that have dependencies on one or more other top level types.
- **Three or more dep**. Similar to above, but with dependencies on three or more other top level types.
- **Five or more dep**. Similar to above.
- **Ten or more dep**. Similar to above. Top level types with this many dependencies will be harder to understand and maintain. So this is measure of how complex the project is.

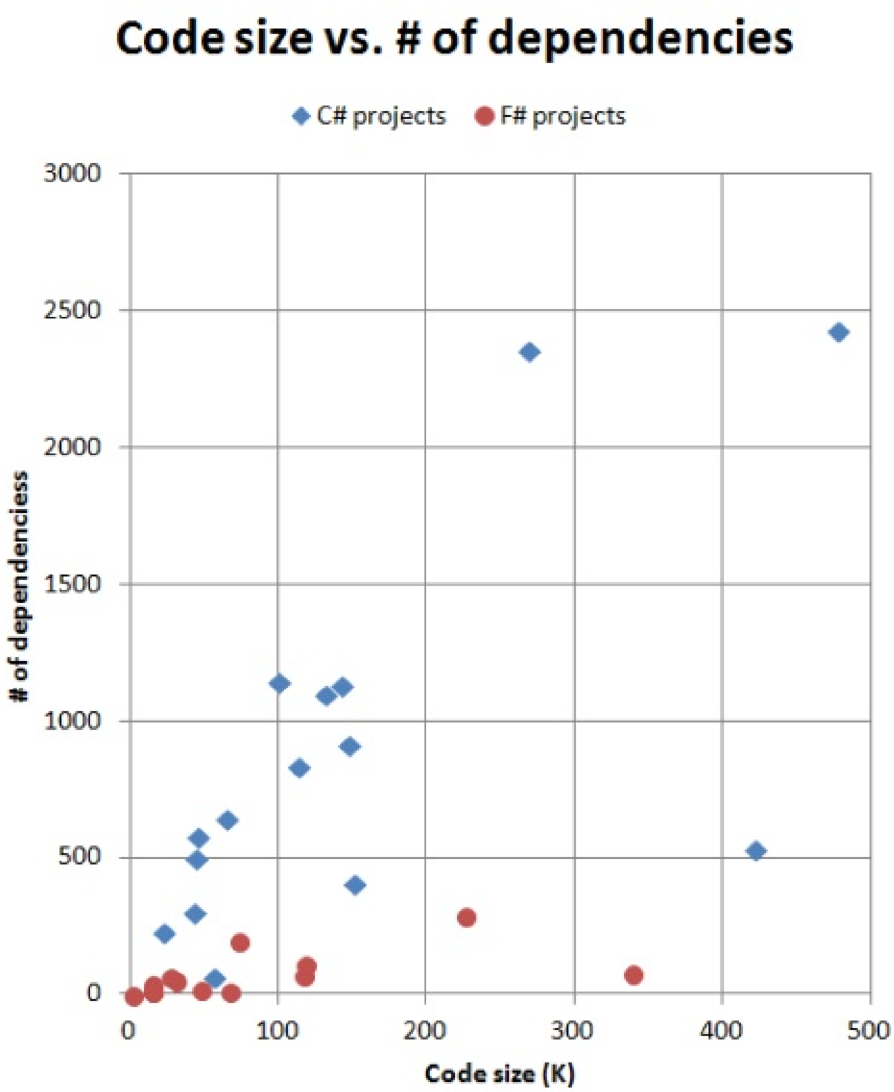
The **diagram** column contains a link to a SVG file, generated from the dependencies, and also the DOT file that was used to generate the SVG. See below for a discussion of these diagrams. (Note that I can't expose the internals of my applications, so I will just give the metrics)

Analysis

These results are very interesting. For C#, the number of total dependencies increases with project size. Each top-level type depends on 3-4 others, on average.

On the other hand, the number of total dependencies in an F# project does not seem to vary too much with project size at all. Each F# module depends on no more than 1-2 others, on average. And the largest project (FSharpX) has a lower ratio than many of the smaller projects. My business app and the Storm project are the only exceptions.

Here's a chart of the relationship between code size and the number of dependencies:



The disparity between C# and F# projects is very clear. The C# dependencies seem to grow linearly with project size, while the F# dependencies seem to be flat.

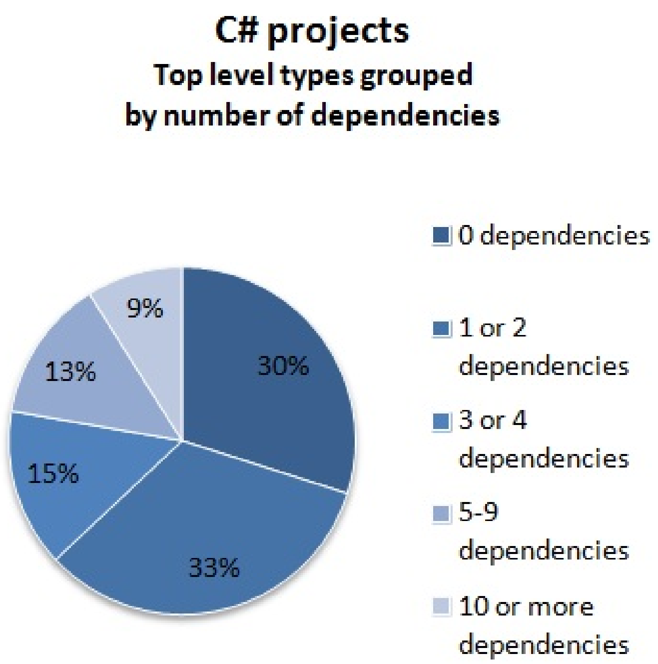
Distribution of dependencies

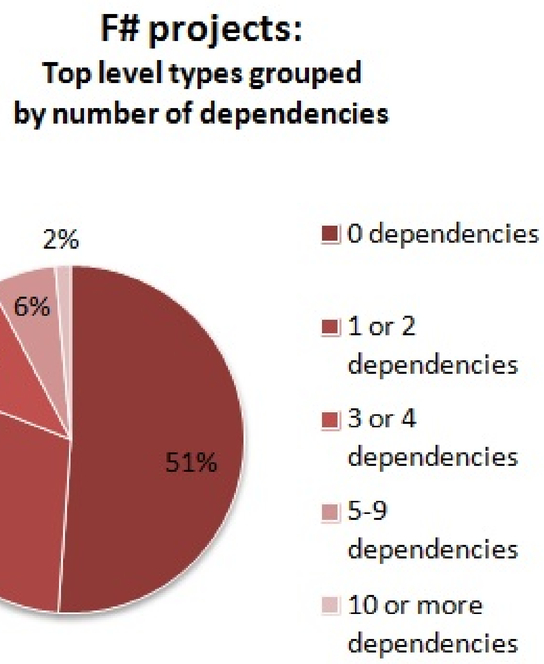
The average number of dependencies per top level type is interesting, but it doesn't help us understand the variability. Are there many modules with lots of dependencies? Or does each one just have a few?

This might make a difference in maintainability, perhaps. I would assume that a module with only one or two dependencies would be easier to understand in the context of the application that one with tens of dependencies.

Rather than doing a sophisticated statistical analysis, I thought I would keep it simple and just count how many top level types had one or more dependencies, three or more dependencies, and so on.

Here are the same results, displayed visually:





So, what can we deduce from these numbers?

- First, in the F# projects, more than half of the modules have no outside dependencies *at all*. This is a bit surprising, but I think it is due to the heavy use of generics compared with C# projects.
- Second, the modules in the F# projects consistently have fewer dependencies than the classes in the C# projects.
- Finally, in the F# projects, modules with a high number of dependencies are quite rare -- less than 2% overall. But in the C# projects, 9% of classes have more than 10 dependencies on other classes.

The worst offender in the F# group is my very own F# application, which is even worse than my C# application with respect to these metrics. Again, it might be due to heavy use of non-generics in the form of domain-specific types, or it might just be that the code needs more refactoring!

The dependency diagrams

It might be useful to look at the dependency diagrams now. These are SVG files, so you should be able to view them in your browser.

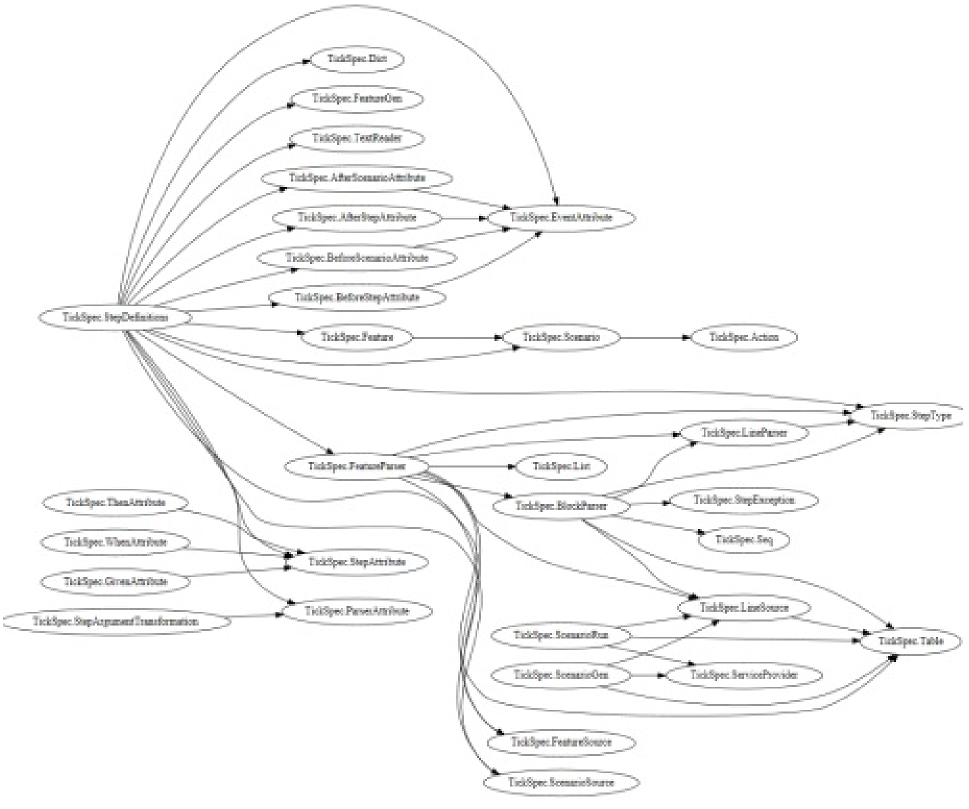
Note that most of these diagrams are very big -- so after you open them you will need to zoom out quite a bit in order to see anything!

Let's start by comparing the diagrams for SpecFlow and TickSpec.

Here's the one for SpecFlow:



Here's the one for TickSpec:



Each diagram lists all the top-level types found in the project. If there is a dependency from one type to another, it is shown by an arrow. The dependencies point from left to right where possible, so any arrows going from right to left implies that there is a cyclic dependency.

The layout is done automatically by graphviz, but in general, the types are organized into columns or "ranks". For example, the SpecFlow diagram has 12 ranks, and the TickSpec diagram has five.

As you can see, there are generally a lot of tangled lines in a typical dependency diagram! How tangled the diagram looks is a sort of visual measure of the code complexity. For instance, if I was tasked to maintain the SpecFlow project, I wouldn't really feel comfortable until I had understood all the relationships between the classes. And the more complex the project, the longer it takes to come up to speed.

OO vs functional design revealed?

The TickSpec diagram is a lot simpler than the SpecFlow one. Is that because TickSpec perhaps doesn't do as much as SpecFlow?

The answer is no, I don't think that it has anything to do with the size of the feature set at all, but rather because the code is organized differently.

Looking at the SpecFlow classes (dotfile), we can see it follows good OOD and TDD practices by creating interfaces. There's a `TestRunnerManager` and an `ITestRunnerManager` , for example. And there are many other patterns that commonly crop up in OOD: "listener" classes and interfaces, "provider" classes and interfaces, "comparer" classes and interfaces, and so on.

But if we look at the TickSpec modules (dotfile) there are no interfaces at all. And no "listeners", "providers" or "comparers" either. There might well be a need for such things in the code, but either they are not exposed outside their module, or more likely, the role they play is fulfilled by functions rather than types.

I'm not picking on the SpecFlow code, by the way. It seems well designed, and is a very useful library, but I think it does highlight some of the differences between OO design and functional design.

Moq compared with Foq

Let's also compare the diagrams for Moq and Foq. These two projects do roughly the same thing, so the code should be comparable.

As before, the project written in F# has a much smaller dependency diagram.

Looking at the Moq classes (dotfile), we can see it includes the "Castle" library, which I didn't eliminate from the analysis. Out of the 249 classes with dependencies, only 66 are Moq specific. If we had considered only the classes in the Moq namespace, we might have had a cleaner diagram.

On the other hand, looking at the Foq modules (dotfile) there are only 23 modules with dependencies, fewer even than just the Moq classes alone.

So something is very different with code organization in F#.

FParsec compared with FParsecCS

The FParsec project is an interesting natural experiment. The project has two assemblies, roughly the same size, but one is written in C# and the other in F#.

It is a bit unfair to compare them directly, because the C# code is designed for parsing fast, while the F# code is more high level. But... I'm going to be unfair and compare them anyway!

Here are the diagrams for the F# assembly "FParsec" and C# assembly "FParsecCS".

They are both nice and clear. Lovely code!

What's not clear from the diagram is that my methodology is being unfair to the C# assembly.

For example, the C# diagram shows that there are dependencies between `Operator`, `OperatorType`, `InfixOperator` and so on. But in fact, looking at the source code, these classes are all in the same physical file. In F#, they would all be in the same module, and their relationships would not count as public dependencies. So the C# code is being penalized in a way.

Even so, looking at the source code, the C# code has 20 source files compared to F#'s 8, so there is still some difference in complexity.

What counts as a dependency?

In defence of my method though, the only thing that is keeping these FParsec C# classes together in the same file is good coding practice; it is not enforced by the C# compiler. Another maintainer could come along and unwittingly separate them into different files, which really *would* increase the complexity. In F# you could not do that so easily, and certainly not accidentally.

So it depends on what you mean by "module", and "dependency". In my view, a module contains things that really are "joined at the hip" and shouldn't easily be decoupled. Hence dependencies within a module don't count, while dependencies between modules do.

Another way to think about it is that F# encourages high coupling in some areas (modules) in exchange for low coupling in others. In C#, the only kind of strict coupling available is class-based. Anything looser, such as using namespaces, has to be enforced using good practices or a tool such as NDepend.

Whether the F# approach is better or worse depends on your preference. It does make certain kinds of refactoring harder as a result.

Cyclic dependencies

Finally, we can turn our attention to the oh-so-evil cyclic dependencies. (If you want to know why they are bad, read this post).

Here are the cyclic dependency results for the C# projects.

Project	Top-level types	Cycle count	Partic.	Partic.%	Max comp. size	Cycle count (public)	Partic. (public)
ef	514	14	123	24%	79	1	7
jsonDotNet	215	3	88	41%	83	1	11
nancy	339	6	35	10%	21	2	4
cecil	240	2	125	52%	123	1	50
nuget	216	4	24	11%	10	0	0
signalR	192	3	14	7%	7	1	5
nunit	173	2	80	46%	78	1	48
specFlow	242	5	11	5%	3	1	2
elmah	116	2	9	8%	5	1	2
yamlDotNet	70	0	0	0%	1	0	0
fparsecCS	41	3	6	15%	2	1	2
moq	397	9	50	13%	15	0	0
ndepend	734	12	79	11%	22	8	36
ndependPlat	185	2	5	3%	3	0	0
personalCS	195	11	34	17%	8	5	19
TOTAL	3869		683	18%			186

And here are the results for the F# projects:

Project	Top-level types	Cycle count	Partic.	Partic.%	Max comp. size	Cycle count (public)	Par (pub
fsxCore	173	0	0	0%	1	0	0
fsCore	154	2	5	3%	3	0	0
fsPowerPack	93	1	2	2%	2	0	0
storm	67	0	0	0%	1	0	0
fParsec	8	0	0	0%	1	0	0
websharper	52	0	0	0%	1	0	0
tickSpec	34	0	0	0%	1	0	0
websharperHtml	18	0	0	0%	1	0	0
canopy	6	0	0	0%	1	0	0
fsYaml	7	0	0	0%	1	0	0
fsSql	13	0	0	0%	1	0	0
fsUnit	2	0	0	0%	0	0	0
foq	35	0	0	0%	1	0	0
personalFS	30	0	0	0%	1	0	0
TOTAL	692		7	1%			0

The columns are:

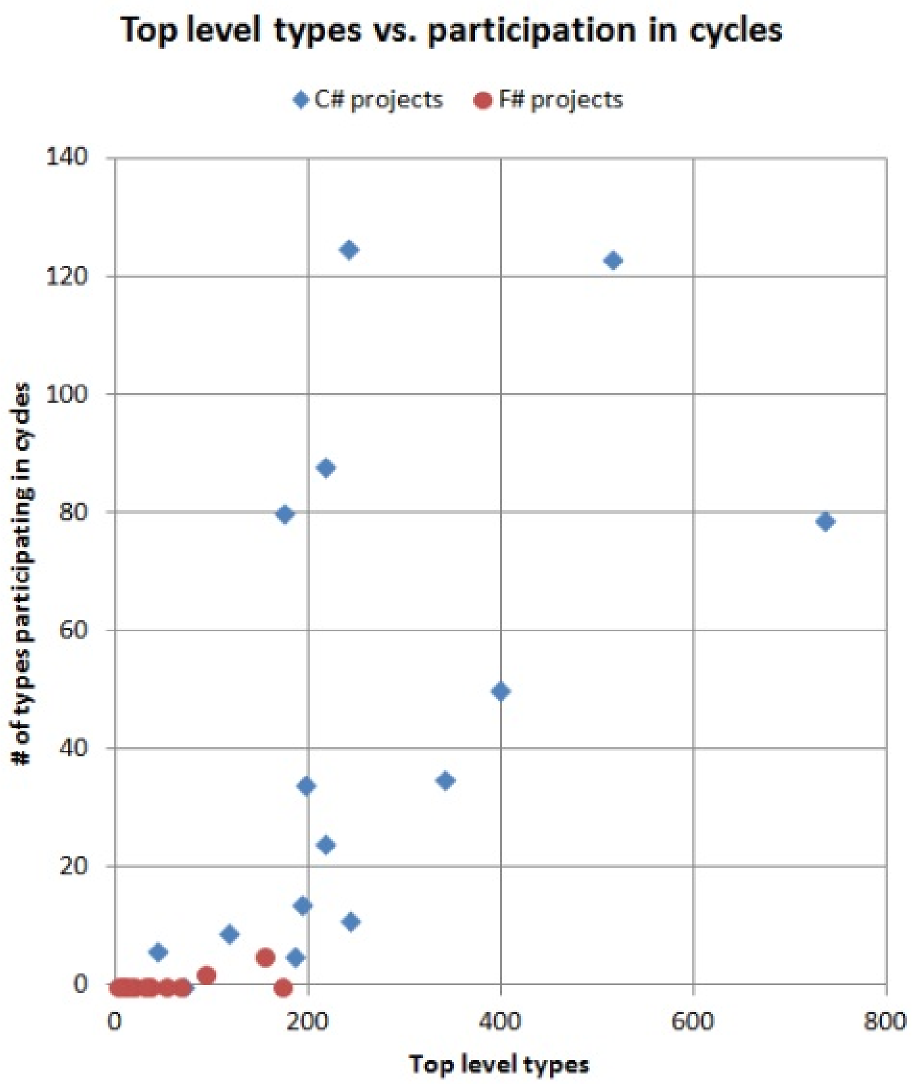
- **Top-level types** is the total number of top-level types in the assembly, as before.
- **Cycle count** is the number of cycles altogether. Ideally it would be zero. But larger is not necessarily worse. Better to have 10 small cycles than one giant one, I think.
- **Partic.**. The number of top level types that participate in any cycle.
- **Partic.%**. The number of top level types that participate in any cycle, as a percent of all types.
- **Max comp. size** is the number of top level types in the largest cyclic component. This is a measure of how complex the cycle is. If there are only two mutually dependent types, then the cycle is a lot less complex than, say, 123 mutually dependent types.
- **... (public)** columns have the same definitions, but using only public dependencies. I thought it would be interesting to see what effect it would have to limit the analysis to public dependencies only.
- The **diagram** column contains a link to a SVG file, generated from the dependencies in the cycles only, and also the DOT file that was used to generate the SVG. See below for an analysis.

Analysis

If we are looking for cycles in the F# code, we will be sorely disappointed. Only two of the F# projects have cycles at all, and those are tiny. For example in FSharp.Core there is a mutual dependency between two types right next to each other in the same file, here.

On the other hand, almost all the C# projects have one or more cycles. Entity Framework has the most cycles, involving 24% of the classes, and Cecil has the worst participation rate, with over half of the classes being involved in a cycle.

Even NDepend has cycles, although to be fair, there may be good reasons for this. First NDepend focuses on removing cycles between namespaces, not classes so much, and second, it's possible that the cycles are between types declared in the same source file. As a result, my method may penalize well-organized C# code somewhat (as noted in the FParsec vs. FParsecCS discussion above).



Why the difference between C# and F#?

- In C#, there is nothing stopping you from creating cycles -- a perfect example of accidental complexity. In fact, you have to make a special effort to avoid them.
- In F#, of course, it is the other way around. You can't easily create cycles at all.

My business applications compared

One more comparison. As part of my day job, I have written a number of business applications in C#, and more recently, in F#. Unlike the other projects listed here, they are very focused on addressing a particular business need, with lots of domain specific code, custom business rules, special cases, and so on.

Both projects were produced under deadline, with changing requirements and all the usual real world constraints that stop you writing ideal code. Like most developers in my position, I would love a chance to tidy them up and refactor them, but they do work, the business is happy, and I have to move on to new things.

Anyway, let's see how they stack up to each other. I can't reveal any details of the code other than the metrics, but I think that should be enough to be useful.

Taking the C# project first:

- It has 195 top level types, about 1 for every 2K of code. Comparing this with other C# projects, there should be *many more* top level types than this. And in fact, I know that this is true. As with many projects (this one is 6 years old) it is lower risk to just add a method to an existing class rather than refactoring it, especially under deadline. Keeping old code stable is always a higher priority than making it beautiful! The result is that classes grow too large over time.
- The flip side of having large classes is that there many fewer cross-class dependencies! It has some of the better scores among the C# projects. So it goes to show that dependencies aren't the only metric. There has to be a balance.
- In terms of cyclic dependencies, it's pretty typical for a C# project. There are a number of them (11) but the largest involves only 8 classes.

Now let's look at my F# project:

- It has 30 modules, about 1 for every 4K of code. Comparing this with other F# projects, it's not excessive, but perhaps a bit of refactoring is in order.
    - As an aside, in my experience with maintaining this code, I have noticed that, unlike C# code, I don't feel that I *have* to add cruft to existing modules when feature requests come in. Instead, I find that in many cases, the faster and lower risk way of making changes is simply to create a *new* module and put all the code for a new feature in there. Because the modules have no state, a function can live anywhere -

  - it is not forced to live in the same class. Over time this approach may create its own problems too (COBOL anyone?) but right now, I find it a breath of fresh air.
- The metrics show that there are an unusually large number of "authored" types per module (4.9). As I noted above, I think this is a result of having fine-grained DDD-style design. The code per authored type is in line with the other F# projects, so that implies they are not too big or small.
- Also, as I noted earlier, the inter-module dependencies are the worst of any F# project. I know that there are some API/service functions that depend on almost all the other modules, but this could be a clue that they might need refactoring.
  - However, unlike C# code, I know exactly where to find these problem modules. I can be fairly certain that all these modules are in the top layer of my application and will thus appear at the bottom of the module list in Visual Studio. How can I be so sure? Because...
- In terms of cyclic dependencies, it's pretty typical for a F# project. There aren't any.

Summary

I started this analysis from curiosity -- was there any meaningful difference in the organization of C# and F# projects?

I was quite surprised that the distinction was so clear. Given these metrics, you could certainly predict which language the assembly was written in.

- **Project complexity**. For a given number of instructions, a C# project is likely to have many more top level types (and hence files) than an F# one -- more than double, it seems.
- **Fine-grained types**. For a given number of modules, a C# project is likely to have fewer authored types than an F# one, implying that the types are not as fine-grained as they could be.
- **Dependencies**. In a C# project, the number of dependencies between classes increases linearly with the size of the project. In an F# project, the number of dependencies is much smaller and stays relatively flat.
- **Cycles**. In a C# project, cycles occur easily unless care is taken to avoid them. In an F# project, cycles are extremely rare, and if present, are very small.

Perhaps this has do with the competency of the programmer, rather than a difference between languages? Well, first of all, I think that the quality of the C# projects is quite good on the whole -- I certainly wouldn't claim that I could write better code! And, in two cases in particular, the C# and F# projects were written by the same person, and differences were still apparent, so I don't think this argument holds up.

Future work

This approach of using *just* the binaries might have gone as far as it can go. For a more accurate analysis, we would need to use metrics from the source code as well (or maybe the pdb file).

For example, a high "instructions per type" metric is good if it corresponds to small source files (concise code), but not if it corresponds to large ones (bloated classes). Similarly, my definition of modularity used top-level types rather than source files, which penalized C# somewhat over F#.

So, I don't claim that this analysis is perfect (and I hope haven't made a terrible mistake in the analysis code!) but I think that it could be a useful starting point for further investigation.

Update 2013-06-15

This post caused quite a bit of interest. Based on feedback, I made the following changes:

Assemblies profiled

- Added Foq and Moq (at the request of Phil Trelford).
- Added the C# component of FParsec (at the request of Dave Thomas and others).
- Added two NDepend assemblies.
- Added two of my own projects, one C# and one F#.

As you can see, adding seven new data points (five C# and two F# projects) didn't change the overall analysis.

Algorithm changes

- Made definition of "authored" type stricter. Excluded types with "GeneratedCodeAttribute" and F# types that are subtypes of a sum type. This had an effect on the F# projects and reduced the "Auth/Top" ratio somewhat.

Text changes

- Rewrote some of the analysis.
- Removed the unfair comparison of YamlDotNet with FParsec.
- Added a comparison of the C# component and F# components of FParsec.
- Added a comparison of Moq and Foq.
- Added a comparison of my own two projects.

The orginal post is still available here

Do you want to port C# code to F#? In this series of posts we'll look at various approaches to this, and the design decisions and trade-offs involved.

- Porting from C# to F#: Introduction. Three approaches to porting existing C# code to F#.
- Getting started with direct porting. F# equivalents to C#.

Porting from C# to F#: Introduction

NOTE: Before reading this series, I suggest that you read the following series as a prerequisite: "thinking functionally", "expressions and syntax", and "understanding F# types".

For many developers, the next step after learning a new language might be to port some existing code over to it, so that they can get a good feel for the differences between the two languages.

As we pointed out earlier, functional languages are very different from imperative languages, and so trying to do a direct port of imperative code to a functional language is often not possible, and even if a crude port is done successfully, the ported code will probably not be using the functional model to its best advantage.

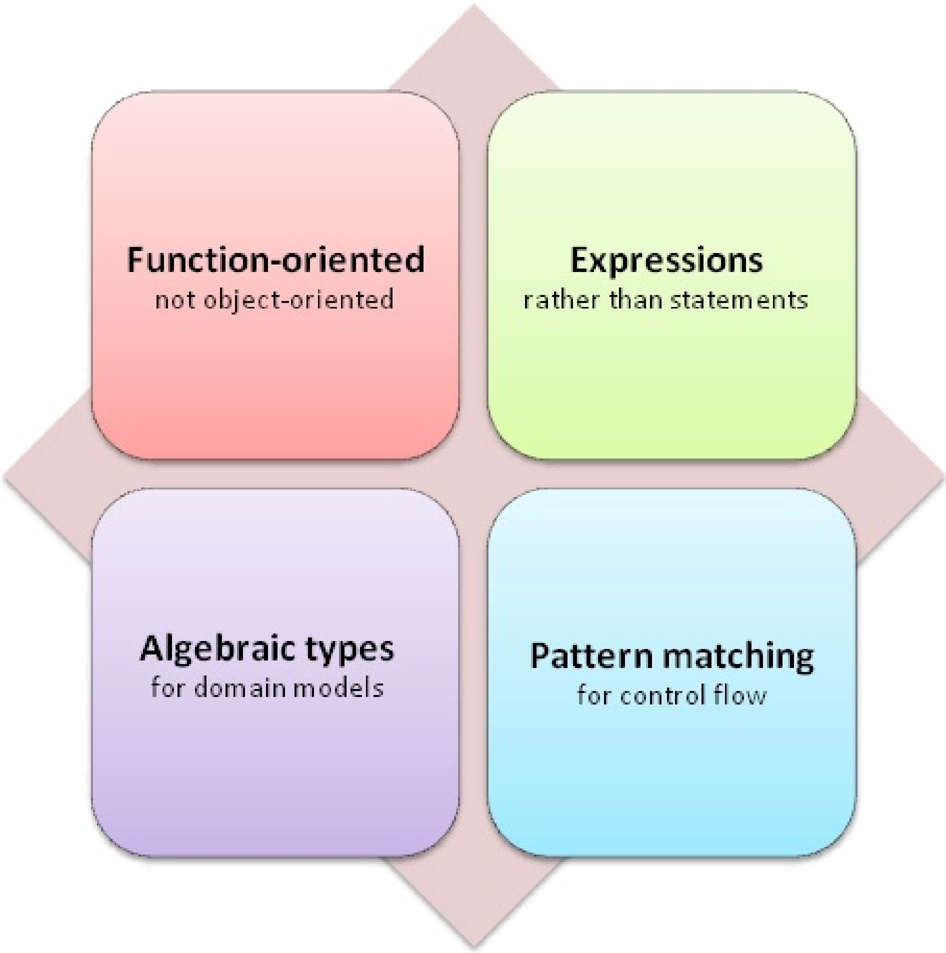
Of course, F# is a multi-paradigm language, and includes support for object-oriented and imperative techniques, but even so, a direct port will generally not be the best way to write the corresponding F# code.

So, in this series, we'll look at various approaches to porting existing C# code to F#.

Levels of porting sophistication

If you recall the diagram from an earlier post, there are four key concepts that differentiate F# from C#.

- Function-oriented rather than object-oriented
- Expressions rather than statements
- Algebraic types for creating domain models
- Pattern matching for flow of control



And, as explained in that post and its sequels, these aspects are not just academic, but offer concrete benefits to you as a developer.

So I have divided the porting process into three levels of sophistication (for lack of a better term), which represent how well the ported code exploits these benefits.

Basic Level: Direct port

At this first level, the F# code is a direct port (where possible) of the C# code. Classes and methods are used instead of modules and functions, and values are frequently mutated.

Intermediate Level: Functional code

At the next level, the F# code has been refactored to be fully functional.

- Classes and methods have been replaced by modules and functions, and values are generally immutable.
- Higher order functions are used to replace interfaces and inheritance.
- Pattern matching is used extensively for control flow.
- Loops have been replaced with list functions such as "map" or recursion.

There are two different paths that can get you to this level.

- The first path is to do a basic direct port to F#, and then refactor the F# code.
- The second path is to convert the existing imperative code to functional code while staying in C#, and only then port the functional C# code to functional F# code!

The second option might seem clumsy, but for real code it will probably be both faster and more comfortable. Faster because you can use a tool such as Resharper to do the refactoring, and more comfortable because you are working in C# until the final port. This approach also makes it clear that the hard part is not the actual port from C# to F#, but the conversion of imperative code to functional code!

Advanced Level: Types represent the domain

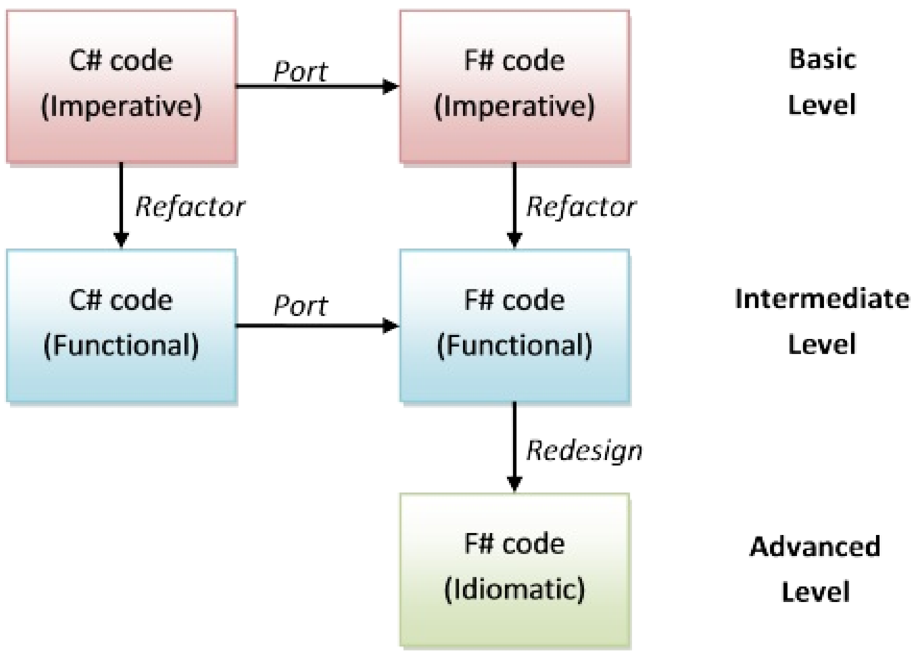
At this final level, not only is the code functional, but the design itself has been changed to exploit the power of algebraic data types (especially union types).

The domain will have been encoded into types such that illegal states are not even representable, and correctness is enforced at compile time. For a concrete demonstration of the power of this approach, see the shopping cart example in the "why use F#" series and the whole "Designing with types" series.

This level can only be done in F#, and is not really practical in C#.

Porting diagram

Here is a diagram to help you visualize the various porting paths described above.



The approach for this series

To see how these three levels work in practice, we'll apply them to some worked examples:

- The first example is a simple system for creating and scoring a ten-pin bowling game, based on the code from the well known "bowling game kata" described by "Uncle" Bob Martin. The original C# code has only one class and about 70 lines of code, but even so, it demonstrates a number of important principles.
- Next, we'll look at some shopping cart code, based on this example.
- The final example is code that represents states for a subway turnstile system, also based on an example from Bob Martin. This example demonstrates how the union types in F# can represent a state transition model more easily than the OO approach.

But first, before we get started on the detailed examples, we'll go back to basics and do some simple porting of some code snippets. That will be the topic of the next post.

Getting started with direct porting

Before we get started on the detailed examples, we'll go back to basics and do some simple porting of trivial examples.

In this post and the next, we'll look at the nearest F# equivalents to common C# statements and keywords, to guide you when doing direct ports.

Basic syntax conversion guidelines

Before starting a port, you need to understand how F# syntax is different from C# syntax. This section presents some general guidelines for converting from one to another. (For a quick overview of F# syntax as a whole, see "F# syntax in 60 seconds")

Curly braces and indentation

C# uses curly braces to indicate the start and end of a block of code. F# generally just uses indentation.

Curly braces are used in F#, but not for blocks of code. Instead, you will see them used:

- For definitions and usage of "record" types.
- In conjunction with computation expressions, such as `seq` and `async`. In general, you will not be using these expressions for basic ports anyway.

For details on the indentation rules, see this post.

Semicolons

Unlike C#'s semicolon, F# does not require any kind of line or statement terminator.

Commas

F# does not use commas for separating parameters or list elements, so remember not to use commas when porting!

For separating list elements, use semicolons rather than commas.

```
// C# example
var list = new int[] { 1,2,3}
```

```
// F# example
let list = [1;2;3] // semicolons
```

For separating parameters for native F# functions, use white space.

```
// C# example
int myFunc(int x, int y, int z) { ... function body ...}
```

```
// F# example
let myFunc (x:int) (y:int) (z:int) :int = ... function body ...
let myFunc x y z = ... function body ...
```

Commas are generally only used for tuples, or for separating parameters when calling .NET library functions. (See this post for more on tuples vs multiple parameters)

Defining variables, functions and types

In F#, definitions of both variables and functions use the form:

```
let someName = // the definition
```

Definitions for all types (classes, structures, interfaces, etc.) use the form:

```
type someName = // the definition
```

The use of the `=` sign is an important difference between F# and C#. Where C# uses curly braces, F# uses the `=` and then the following block of code must be indented.

Mutable values

In F#, values are immutable by default. If you are doing a direct imperative port, you probably need to make some of the values mutable, using the `mutable` keyword. Then to assign to the values, use the `<-` operator, not the equals sign.

```
// C# example
var variableName = 42
variableName = variableName + 1
```

```
// F# example
let mutable variableName = 42
variableName <- variableName + 1
```

Assignment vs. testing for equality

In C#, the equals sign is used for assignment, and the double equals `==` is used for testing equality.

However in F#, the equals sign is used for testing equality, and is also used to initially bind values to other values when declared,

```
let mutable variableName = 42     // Bound to 42 on declaration
variableName <- variableName + 1  // Mutated (reassigned)
variableName = variableName + 1   // Comparison not assignment!
```

To test for inequality, use SQL-style `<>` rather than `!=`

```
let variableName = 42             // Bound to 42 on declaration
variableName <> 43                // Comparison will return true.
variableName != 43                // Error FS0020.
```

If you accidentally use `!=` you will probably get an error FS0020.

Conversion example #1

With these basic guidelines in place, let's look at some real code examples, and do a direct port for them.

This first example has some very simple code, which we will port line by line. Here's the C# code.

```
using System;
using System.Collections.Generic;

namespace PortingToFsharp
{
    public class Squarer
    {
        public int Square(int input)
        {
            var result = input * input;
            return result;
        }

        public void PrintSquare(int input)
        {
            var result = this.Square(input);
            Console.WriteLine("Input={0}. Result={1}",
              input, result);
        }
    }
```

Converting "using" and "namespace"

These keywords are straightforward:

- `using` becomes `open`
- `namespace` with curly braces becomes just `namespace`.

Unlike C#, F# files do not generally declare namespaces unless they need to interop with other .NET code. The filename itself acts as a default namespace.

Note that the namespace, if used, must come before anything else, such as "open". This the opposite order from most C# code.

Converting the class

To declare a simple class, use:

```
type myClassName() =
   ... code ...
```

Note that there are parentheses after the class name. These are required for class definitions.

More complicated class definitions will be shown in the next example, and you read the complete discussion of classes.

Converting function/method signatures

For function/method signatures:

- Parentheses are not needed around the parameter list
- Whitespace is used to separate the parameters, not commas
- Rather than curly braces, an equals sign signals the start of the function body
- The parameters don't normally need types but if you do need them:
  - The type name comes after the value or parameter
  - The parameter name and type are separated by colons
  - When specifying types for parameters, you should probably wrap the pair in parentheses to avoid unexpected behavior.
  - The return type for the function as a whole is prefixed by a colon, and comes after all the other parameters

Here's a C# function signature:

```
int Square(int input) { ... code ...}
```

and here's the corresponding F# function signature with explicit types:

```
let Square (input:int) :int =  ... code ...
```

However, because F# can normally infer the parameter and return types, you rarely need to specify them explicitly.

Here's a more typical F# signature, with inferred types:

```
let Square input =  ... code ...
```

void

The `void` keyword in C# is generally not needed, but if required, would be converted to `unit`

So the C# code:

```
void PrintSquare(int input) { ... code ...}
```

could be converted to the F# code:

```
let PrintSquare (input:int) :unit =  ... code ...
```

but again, the specific types are rarely needed, and so the F# version is just:

```
let PrintSquare input =  ... code ...
```

Converting function/method bodies

In a function body, you are likely to have a combination of:

- Variable declarations and assignments
- Function calls
- Control flow statements
- Return values

We'll have a quick look at porting each of these in turn, except for control flow, which we'll discuss later.

Converting variable declarations

Almost always, you can use `let` on its own, just like `var` in C#:

```
// C# variable declaration
var result = input * input;
```

```
// F# value declaration
let result = input * input
```

Unlike C#, you must always assign ("bind") something to an F# value as part of its declaration.

```
// C# example
int unassignedVariable; //valid
```

```
// F# example
let unassignedVariable // not valid
```

As noted above, if you need to change the value after its declaration, you must use the "mutable" keyword.

If you need to specify a type for a value, the type name comes after the value or parameter, preceded by a colon.

```
// C# example
int variableName = 42;
```

```
// F# example
let variableName:int = 42
```

Converting function calls

When calling a native F# function, there is no need for parentheses or commas. In other words, the same rules apply for calling a function as when defining it.

Here's C# code for defining a function, then calling it:

```
// define a method/function
int Square(int input) { ... code  ...}

// call it
var result = Square(input);
```

However, because F# can normally infer the parameter and return types, you rarely need to specify them explicitly So here's typical F# code for defining a function and then calling it:

```
// define a function
let Square input = ... code ...

// call it
let result = Square input
```

Return values

In C#, you use the `return` keyword. But in F#, the last value in the block is automatically the "return" value.

Here's the C# code returning the `result` variable.

```
public int Square(int input)
{
    var result = input * input;
    return result;   //explicit "return" keyword
}
```

And here's the F# equivalent.

```
let Square input =
    let result = input * input
    result        // implicit "return" value
```

This is because F# is expression-based. Everything is an expression, and the value of a block expression as a whole is just the value of the last expression in the block.

For more details on expression-oriented code, see "expressions vs statements".

Printing to the console

To print output in C#, you generally use `Console.WriteLine` or similar. In F#, you generally use `printf` or similar, which is typesafe. (More details on using "printf" family).

The complete port of example #1

Putting it all together, here is the complete direct port of example #1 to F#.

The C# code again:

```
using System;
using System.Collections.Generic;

namespace PortingToFsharp
{
    public class Squarer
    {
        public int Square(int input)
        {
            var result = input * input;
            return result;
        }

        public void PrintSquare(int input)
        {
            var result = this.Square(input);
            Console.WriteLine("Input={0}. Result={1}",
              input, result);
        }
    }
```

And the equivalent F# code:

```
namespace PortingToFsharp

open System
open System.Collections.Generic

type Squarer() =

    let Square input =
        let result = input * input
        result

    let PrintSquare input =
        let result = Square input
        printf "Input=%i. Result=%i" input result
```

In this series, we'll look at some of the ways we can use types as part of the design process. In particular, the thoughtful use of types can make a design more transparent and improve correctness at the same time.

This series will be focused on the "micro level" of design. That is, working at the lowest level of individual types and functions. Higher level design approaches, and the associated decisions about using functional or object-oriented style, will be discussed in another series.

Many of the suggestions are also feasable in C# or Java, but the lightweight nature of F# types means that it is much more likely that we will do this kind of refactoring.

- Designing with types: Introduction. Making design more transparent and improving correctness.
- Designing with types: Single case union types. Adding meaning to primitive types.
- Designing with types: Making illegal states unrepresentable. Encoding business logic in types.
- Designing with types: Discovering new concepts. Gaining deeper insight into the domain.
- Designing with types: Making state explicit. Using state machines to ensure correctness.
- Designing with types: Constrained strings. Adding more semantic information to a primitive type.
- Designing with types: Non-string types. Working with integers and dates safely.
- Designing with types: Conclusion. A before and after comparison.

Designing with types: Introduction

In this series, we'll look at some of the ways we can use types as part of the design process. In particular, the thoughtful use of types can make a design more transparent and improve correctness at the same time.

This series will be focused on the "micro level" of design. That is, working at the lowest level of individual types and functions. Higher level design approaches, and the associated decisions about using functional or object-oriented style, will be discussed in another series.

Many of the suggestions are also feasable in C# or Java, but the lightweight nature of F# types means that it is much more likely that we will do this kind of refactoring.

A basic example

For demonstration of the various uses of types, I'll work with a very straightforward example, namely a `Contact` type, such as the one below.

```
type Contact =
    {
    FirstName: string;
    MiddleInitial: string;
    LastName: string;

    EmailAddress: string;
    //true if ownership of email address is confirmed
    IsEmailVerified: bool;

    Address1: string;
    Address2: string;
    City: string;
    State: string;
    Zip: string;
    //true if validated against address service
    IsAddressValid: bool;
    }
```

This seems very obvious -- I'm sure we have all seen something like this many times. So what can we do with it? How can we refactor this to make the most of the type system?

Creating "atomic" types

The first thing to do is to look at the usage pattern of data access and updates. For example, would be it be likely that `Zip` is updated without also updating `Address1` at the same time? On the other hand, it might be common that a transaction updates `EmailAddress` but not `FirstName`.

This leads to the first guideline:

- *Guideline: Use records or tuples to group together data that are required to be consistent (that is "atomic") but don't needlessly group together data that is not related.*

In this case, it is fairly obvious that the three name values are a set, the address values are a set, and the email is also a set.

We have also some extra flags here, such as `IsAddressValid` and `IsEmailVerified`. Should these be part of the related set or not? Certainly yes for now, because the flags are dependent on the related values.

For example, if the `EmailAddress` changes, then `IsEmailVerified` probably needs to be reset to false at the same time.

For `PostalAddress`, it seems clear that the core "address" part is a useful common type, without the `IsAddressValid` flag. On the other hand, the `IsAddressValid` is associated with the address, and will be updated when it changes.

So it seems that we should create *two* types. One is a generic `PostalAddress` and the other is an address in the context of a contact, which we can call `PostalContactInfo`, say.

```
type PostalAddress =
    {
    Address1: string;
    Address2: string;
    City: string;
    State: string;
    Zip: string;
    }

type PostalContactInfo =
    {
    Address: PostalAddress;
    IsAddressValid: bool;
    }
```

Finally, we can use the option type to signal that certain values, such as `MiddleInitial`, are indeed optional.

```
type PersonalName =
    {
    FirstName: string;
    // use "option" to signal optionality
    MiddleInitial: string option;
    LastName: string;
    }
```

Summary

With all these changes, we now have the following code:

```
type PersonalName =
    {
    FirstName: string;
    // use "option" to signal optionality
    MiddleInitial: string option;
    LastName: string;
    }

type EmailContactInfo =
    {
    EmailAddress: string;
    IsEmailVerified: bool;
    }

type PostalAddress =
    {
    Address1: string;
    Address2: string;
    City: string;
    State: string;
    Zip: string;
    }

type PostalContactInfo =
    {
    Address: PostalAddress;
    IsAddressValid: bool;
    }

type Contact =
    {
    Name: PersonalName;
    EmailContactInfo: EmailContactInfo;
    PostalContactInfo: PostalContactInfo;
    }
```

We haven't written a single function yet, but already the code represents the domain better. However, this is just the beginning of what we can do.

Next up, using single case unions to add semantic meaning to primitive types.

Designing with types: Single case union types

At the end of the previous post, we had values for email addresses, zip codes, etc., defined like this:

```
EmailAddress: string;
State: string;
Zip: string;
```

These are all defined as simple strings. But really, are they just strings? Is an email address interchangable with a zip code or a state abbreviation?

In a domain driven design, they are indeed distinct things, not just strings. So we would ideally like to have lots of separate types for them so that they cannot accidentally be mixed up.

This has been known as good practice for a long time, but in languages like C# and Java it can be painful to create hundred of tiny types like this, leading to the so called "primitive obsession" code smell.

But F# there is no excuse! It is trivial to create simple wrapper types.

Wrapping primitive types

The simplest way to create a separate type is to wrap the underlying string type inside another type.

We can do it using single case union types, like so:

```
type EmailAddress = EmailAddress of string
type ZipCode = ZipCode of string
type StateCode = StateCode of string
```

or alternatively, we could use record types with one field, like this:

```
type EmailAddress = { EmailAddress: string }
type ZipCode = { ZipCode: string }
type StateCode = { StateCode: string}
```

Both approaches can be used to create wrapper types around a string or other primitive type, so which way is better?

The answer is generally the single case discriminated union. It is much easier to "wrap" and "unwrap", as the "union case" is actually a proper constructor function in its own right. Unwrapping can be done using inline pattern matching.

Here's some examples of how an `EmailAddress` type might be constructed and deconstructed:

```
type EmailAddress = EmailAddress of string

// using the constructor as a function
"a" |> EmailAddress
["a"; "b"; "c"] |> List.map EmailAddress

// inline deconstruction
let a' = "a" |> EmailAddress
let (EmailAddress a'') = a'

let addresses =
    ["a"; "b"; "c"]
    |> List.map EmailAddress

let addresses' =
    addresses
    |> List.map (fun (EmailAddress e) -> e)
```

You can't do this as easily using record types.

So, let's refactor the code again to use these union types. It now looks like this:

```
type PersonalName =
    {
    FirstName: string;
    MiddleInitial: string option;
    LastName: string;
    }

type EmailAddress = EmailAddress of string

type EmailContactInfo =
    {
    EmailAddress: EmailAddress;
    IsEmailVerified: bool;
    }

type ZipCode = ZipCode of string
type StateCode = StateCode of string

type PostalAddress =
    {
    Address1: string;
    Address2: string;
    City: string;
    State: StateCode;
    Zip: ZipCode;
    }

type PostalContactInfo =
    {
    Address: PostalAddress;
    IsAddressValid: bool;
    }

type Contact =
    {
    Name: PersonalName;
    EmailContactInfo: EmailContactInfo;
    PostalContactInfo: PostalContactInfo;
    }
```

Another nice thing about the union type is that the implementation can be encapsulated with module signatures, as we'll discuss below.

Naming the "case" of a single case union

In the examples above we used the same name for the case as we did for the type:

```
type EmailAddress = EmailAddress of string
type ZipCode = ZipCode of string
type StateCode = StateCode of string
```

This might seem confusing initially, but really they are in different scopes, so there is no naming collision. One is a type, and one is a constructor function with the same name.

So if you see a function signature like this:

```
val f: string -> EmailAddress
```

this refers to things in the world of types, so `EmailAddress` refers to the type.

On the other hand, if you see some code like this:

```
let x = EmailAddress y
```

this refers to things in the world of values, so `EmailAddress` refers to the constructor function.

Constructing single case unions

For values that have special meaning, such as email addresses and zip codes, generally only certain values are allowed. Not every string is an acceptable email or zip code.

This implies that we will need to do validation at some point, and what better point than at construction time? After all, once the value is constructed, it is immutable, so there is no worry that someone might modify it later.

Here's how we might extend the above module with some constructor functions:

```
... types as above ...

let CreateEmailAddress (s:string) =
    if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then Some (EmailAddress s)
        else None

let CreateStateCode (s:string) =
    let s' = s.ToUpper()
    let stateCodes = ["AZ";"CA";"NY"] //etc
    if stateCodes |> List.exists ((=) s')
        then Some (StateCode s')
        else None
```

We can test the constructors now:

```
CreateStateCode "CA"
CreateStateCode "XX"

CreateEmailAddress "a@example.com"
CreateEmailAddress "example.com"
```

Handling invalid input in a constructor

With these kinds of constructor functions, one immediate challenge is the question of how to handle invalid input. For example, what should happen if I pass in "abc" to the email address constructor?

There are a number of ways to deal with it.

First, you could throw an exception. I find this ugly and unimaginative, so I'm rejecting this one out of hand!

Next, you could return an option type, with `None` meaning that the input wasn't valid. This is what the constructor functions above do.

This is generally the easiest approach. It has the advantage that the caller has to explicitly handle the case when the value is not valid.

For example, the caller's code for the example above might look like:

```
match (CreateEmailAddress "a@example.com") with
| Some email -> ... do something with email
| None -> ... ignore?
```

The disadvantage is that with complex validations, it might not be obvious what went wrong. Was the email too long, or missing a '@' sign, or an invalid domain? We can't tell.

If you do need more detail, you might want to return a type which contains a more detailed explanation in the error case.

The following example uses a `CreationResult` type to indicate the error in the failure case.

```
type EmailAddress = EmailAddress of string
type CreationResult<'T> = Success of 'T | Error of string

let CreateEmailAddress2 (s:string) =
    if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then Success (EmailAddress s)
        else Error "Email address must contain an @ sign"

// test
CreateEmailAddress2 "example.com"
```

Finally, the most general approach uses continuations. That is, you pass in two functions, one for the success case (that takes the newly constructed email as parameter), and another for the failure case (that takes the error string as parameter).

```
type EmailAddress = EmailAddress of string

let CreateEmailAddressWithContinuations success failure (s:string) =
    if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then success (EmailAddress s)
        else failure "Email address must contain an @ sign"
```

The success function takes the email as a parameter and the error function takes a string. Both functions must return the same type, but the type is up to you.

Here is a simple example -- both functions do a printf, and return nothing (i.e. unit).

```
let success (EmailAddress s) = printfn "success creating email %s" s
let failure  msg = printfn "error creating email: %s" msg
CreateEmailAddressWithContinuations success failure "example.com"
CreateEmailAddressWithContinuations success failure "x@example.com"
```

With continuations, you can easily reproduce any of the other approaches. Here's the way to create options, for example. In this case both functions return an `EmailAddress option`.

```
let success e = Some e
let failure _  = None
CreateEmailAddressWithContinuations success failure "example.com"
CreateEmailAddressWithContinuations success failure "x@example.com"
```

And here is the way to throw exceptions in the error case:

```
let success e = e
let failure _  = failwith "bad email address"
CreateEmailAddressWithContinuations success failure "example.com"
CreateEmailAddressWithContinuations success failure "x@example.com"
```

This code seems quite cumbersome, but in practice you would probably create a local partially applied function that you use instead of the long-winded one.

```
// setup a partially applied function
let success e = Some e
let failure _  = None
let createEmail = CreateEmailAddressWithContinuations success failure

// use the partially applied function
createEmail "x@example.com"
createEmail "example.com"
```

Creating modules for wrapper types

These simple wrapper types are starting to get more complicated now that we are adding validations, and we will probably discover other functions that we want to associate with the type.

So it is probably a good idea to create a module for each wrapper type, and put the type and its associated functions there.

```
module EmailAddress =

    type T = EmailAddress of string

    // wrap
    let create (s:string) =
        if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
            then Some (EmailAddress s)
            else None

    // unwrap
    let value (EmailAddress e) = e
```

The users of the type would then use the module functions to create and unwrap the type. For example:

```

// create email addresses
let address1 = EmailAddress.create "x@example.com"
let address2 = EmailAddress.create "example.com"

// unwrap an email address
match address1 with
| Some e -> EmailAddress.value e |> printfn "the value is %s"
| None -> ()
```

Forcing use of the constructor

One issue is that you cannot force callers to use the constructor. Someone could just bypass the validation and create the type directly.

In practice, that tends not to be a problem. One simple techinique is to use naming conventions to indicate a "private" type, and provide "wrap" and "unwrap" functions so that the clients never need to interact with the type directly.

Here's an example:

```
module EmailAddress =

    // private type
    type _T = EmailAddress of string

    // wrap
    let create (s:string) =
        if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
            then Some (EmailAddress s)
            else None

    // unwrap
    let value (EmailAddress e) = e
```

Of course the type is not really private in this case, but you are encouraging the callers to always use the "published" functions.

If you really want to encapsulate the internals of the type and force callers to use a constructor function, you can use module signatures.

Here's a signature file for the email address example:

```
// FILE: EmailAddress.fsi

module EmailAddress

// encapsulated type
type T

// wrap
val create : string -> T option

// unwrap
val value : T -> string
```

(Note that module signatures only work in compiled projects, not in interactive scripts, so to test this, you will need to create three files in an F# project, with the filenames as shown here.)

Here's the implementation file:

```
// FILE: EmailAddress.fs

module EmailAddress

// encapsulated type
type T = EmailAddress of string

// wrap
let create (s:string) =
    if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        then Some (EmailAddress s)
        else None

// unwrap
let value (EmailAddress e) = e
```

And here's a client:

```
// FILE: EmailAddressClient.fs

module EmailAddressClient

open EmailAddress

// code works when using the published functions
let address1 = EmailAddress.create "x@example.com"
let address2 = EmailAddress.create "example.com"

// code that uses the internals of the type fails to compile
let address3 = T.EmailAddress "bad email"
```

The type `EmailAddress.T` exported by the module signature is opaque, so clients cannot access the internals.

As you can see, this approach enforces the use of the constructor. Trying to create the type directly (`T.EmailAddress "bad email"`) causes a compile error.

When to "wrap" single case unions

Now that we have the wrapper type, when should we construct them?

Generally you only need to at service boundaries (for example, boundaries in a hexagonal architecture)

In this approach, wrapping is done in the UI layer, or when loading from a persistence layer, and once the wrapped type is created, it is passed in to the domain layer and manipulated "whole", as an opaque type. It is suprisingly uncommon that you actually need the wrapped contents directly when working in the domain itself.

As part of the construction, it is critical that the caller uses the provided constructor rather than doing its own validation logic. This ensures that "bad" values can never enter the domain.

For example, here is some code that shows the UI doing its own validation:

```
let processFormSubmit () =
    let s = uiTextBox.Text
    if (s.Length < 50)
        then // set email on domain object
        else // show validation error message
```

A better way is to let the constructor do it, as shown earlier.

```
let processFormSubmit () =
    let emailOpt = uiTextBox.Text |> EmailAddress.create
    match emailOpt with
    | Some email -> // set email on domain object
    | None -> // show validation error message
```

When to "unwrap" single case unions

And when is unwrapping needed? Again, generally only at service boundaries. For example, when you are persisting an email to a database, or binding to a UI element or view model.

One tip to avoid explicit unwrapping is to use the continuation approach again, passing in a function that will be applied to the wrapped value.

That is, rather than calling the "unwrap" function explicitly:

```
address |> EmailAddress.value |> printfn "the value is %s"
```

You would pass in a function which gets applied to the inner value, like this:

```
address |> EmailAddress.apply (printfn "the value is %s")
```

Putting this together, we now have the complete `EmailAddress` module.

```
module EmailAddress =

    type _T = EmailAddress of string

    // create with continuation
    let createWithCont success failure (s:string) =
        if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
            then success (EmailAddress s)
            else failure "Email address must contain an @ sign"

    // create directly
    let create s =
        let success e = Some e
        let failure _  = None
        createWithCont success failure s

    // unwrap with continuation
    let apply f (EmailAddress e) = f e

    // unwrap directly
    let value e = apply id e
```

The `create` and `value` functions are not strictly necessary, but are added for the convenience of callers.

The code so far

Let's refactor the `Contact` code now, with the new wrapper types and modules added.

```
module EmailAddress =

    type T = EmailAddress of string

    // create with continuation
    let createWithCont success failure (s:string) =
        if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
            then success (EmailAddress s)
            else failure "Email address must contain an @ sign"

    // create directly
    let create s =
        let success e = Some e
        let failure _  = None
        createWithCont success failure s

    // unwrap with continuation
    let apply f (EmailAddress e) = f e
```

```
    // unwrap directly
    let value e = apply id e

module ZipCode =

    type T = ZipCode of string

    // create with continuation
    let createWithCont success failure  (s:string) =
        if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\d{5}$")
            then success (ZipCode s)
            else failure "Zip code must be 5 digits"

    // create directly
    let create s =
        let success e = Some e
        let failure _  = None
        createWithCont success failure s

    // unwrap with continuation
    let apply f (ZipCode e) = f e

    // unwrap directly
    let value e = apply id e

module StateCode =

    type T = StateCode of string

    // create with continuation
    let createWithCont success failure  (s:string) =
        let s' = s.ToUpper()
        let stateCodes = ["AZ";"CA";"NY"] //etc
        if stateCodes |> List.exists ((=) s')
            then success (StateCode s')
            else failure "State is not in list"

    // create directly
    let create s =
        let success e = Some e
        let failure _  = None
        createWithCont success failure s

    // unwrap with continuation
    let apply f (StateCode e) = f e

    // unwrap directly
    let value e = apply id e

type PersonalName =
    {
    FirstName: string;
    MiddleInitial: string option;
```

```
    LastName: string;
    }

type EmailContactInfo =
    {
    EmailAddress: EmailAddress.T;
    IsEmailVerified: bool;
    }

type PostalAddress =
    {
    Address1: string;
    Address2: string;
    City: string;
    State: StateCode.T;
    Zip: ZipCode.T;
    }

type PostalContactInfo =
    {
    Address: PostalAddress;
    IsAddressValid: bool;
    }

type Contact =
    {
    Name: PersonalName;
    EmailContactInfo: EmailContactInfo;
    PostalContactInfo: PostalContactInfo;
    }
```

By the way, notice that we now have quite a lot of duplicate code in the three wrapper type modules. What would be a good way of getting rid of it, or at least making it cleaner?

Summary

To sum up the use of discriminated unions, here are some guidelines:

- Do use single case discriminated unions to create types that represent the domain accurately.
- If the wrapped value needs validation, then provide constructors that do the validation and enforce their use.
- Be clear what happens when validation fails. In simple cases, return option types. In more complex cases, let the caller pass in handlers for success and failure.
- If the wrapped value has many associated functions, consider moving it into its own module.
- If you need to enforce encapsulation, use signature files.

We're still not done with refactoring. We can alter the design of types to enforce business rules at compile time -- making illegal states unrepresentable.

Update

Many people have asked for more information on how to ensure that constrained types such as `EmailAddress` are only created through a special constructor that does the validation. So I have created a gist here that has some detailed examples of other ways of doing it.

Designing with types: Making illegal states unrepresentable

In this post, we look at a key benefit of F#, which using the type system to "make illegal states unrepresentable" (a phrase borrowed from Yaron Minsky).

Let's look at our `Contact` type. Thanks to the previous refactoring, it is quite simple:

```
type Contact =
    {
    Name: Name;
    EmailContactInfo: EmailContactInfo;
    PostalContactInfo: PostalContactInfo;
    }
```

Now let's say that we have the following simple business rule: *"A contact must have an email or a postal address"*. Does our type conform to this rule?

The answer is no. The business rule implies that a contact might have an email address but no postal address, or vice versa. But as it stands, our type requires that a contact must always have *both* pieces of information.

The answer seems obvious -- make the addresses optional, like this:

```
type Contact =
    {
    Name: PersonalName;
    EmailContactInfo: EmailContactInfo option;
    PostalContactInfo: PostalContactInfo option;
    }
```

But now we have gone too far the other way. In this design, it would be possible for a contact to have neither type of address at all. But the business rule says that at least one piece of information *must* be present.

What's the solution?

Making illegal states unrepresentable

If we think about the business rule carefully, we realize that there are three possibilities:

- A contact only has an email address
- A contact only has a postal address
- A contact has both a email address and a postal address

Once it is put like this, the solution becomes obvious -- use a union type with a case for each possibility.

```
type ContactInfo =
    | EmailOnly of EmailContactInfo
    | PostOnly of PostalContactInfo
    | EmailAndPost of EmailContactInfo * PostalContactInfo

type Contact =
    {
    Name: Name;
    ContactInfo: ContactInfo;
    }
```

This design meets the requirements perfectly. All three cases are explictly represented, and the fourth possible case (with no email or postal address at all) is not allowed.

Note that for the "email and post" case, I just used a tuple type for now. It's perfectly adequate for what we need.

Constructing a ContactInfo

Now let's see how we might use this in practice. We'll start by creating a new contact:

```
let contactFromEmail name emailStr =
    let emailOpt = EmailAddress.create emailStr
    // handle cases when email is valid or invalid
    match emailOpt with
    | Some email ->
        let emailContactInfo =
            {EmailAddress=email; IsEmailVerified=false}
        let contactInfo = EmailOnly emailContactInfo
        Some {Name=name; ContactInfo=contactInfo}
    | None -> None

let name = {FirstName = "A"; MiddleInitial=None; LastName="Smith"}
let contactOpt = contactFromEmail name "abc@example.com"
```

In this code, we have created a simple helper function `contactFromEmail` to create a new contact by passing in a name and email. However, the email might not be valid, so the function has to handle both cases, which it doesn't by returning a `Contact option`, not a `Contact`

Updating a ContactInfo

Now if we need to add a postal address to an existing `ContactInfo`, we have no choice but to handle all three possible cases:

- If a contact previously only had an email address, it now has both an email address and a postal address, so return a contact using the `EmailAndPost` case.
- If a contact previously only had a postal address, return a contact using the `PostOnly` case, replacing the existing address.
- If a contact previously had both an email address and a postal address, return a contact with using the `EmailAndPost` case, replacing the existing address.

So here's a helper method that updates the postal address. You can see how it explicitly handles each case.

```
let updatePostalAddress contact newPostalAddress =
    let {Name=name; ContactInfo=contactInfo} = contact
    let newContactInfo =
        match contactInfo with
        | EmailOnly email ->
            EmailAndPost (email,newPostalAddress)
        | PostOnly _ -> // ignore existing address
            PostOnly newPostalAddress
        | EmailAndPost (email,_) -> // ignore existing address
            EmailAndPost (email,newPostalAddress)
    // make a new contact
    {Name=name; ContactInfo=newContactInfo}
```

And here is the code in use:

```
let contact = contactOpt.Value   // see warning about option.Value below
let newPostalAddress =
    let state = StateCode.create "CA"
    let zip = ZipCode.create "97210"
    {
        Address =
            {
            Address1= "123 Main";
            Address2="";
            City="Beverly Hills";
            State=state.Value; // see warning about option.Value below
            Zip=zip.Value;     // see warning about option.Value below
            };
        IsAddressValid=false
    }
let newContact = updatePostalAddress contact newPostalAddress
```

WARNING: I am using `option.Value` to extract the contents of an option in this code. This is ok when playing around interactively but is extremely bad practice in production code! You should always use matching to handle both cases of an option.

Why bother to make these complicated types?

At this point, you might be saying that we have made things unnecessarily complicated. I would answer with these points:

First, the business logic *is* complicated. There is no easy way to avoid it. If your code is not this complicated, you are not handling all the cases properly.

Second, if the logic is represented by types, it is automatically self documenting. You can look at the union cases below and immediate see what the business rule is. You do not have to spend any time trying to analyze any other code.

```
type ContactInfo =
    | EmailOnly of EmailContactInfo
    | PostOnly of PostalContactInfo
    | EmailAndPost of EmailContactInfo * PostalContactInfo
```

Finally, if the logic is represented by a type, any changes to the business rules will immediately create breaking changes, which is a generally a good thing.

In the next post, we'll dig deeper into the last point. As you try to represent business logic using types, you may suddenly find that can gain a whole new insight into the domain.

Designing with types: Discovering new concepts

In the last post, we looked at how we could represent a business rule using types.

The rule was: *"A contact must have an email or a postal address"*.

And the type we designed was:

```
type ContactInfo =
    | EmailOnly of EmailContactInfo
    | PostOnly of PostalContactInfo
    | EmailAndPost of EmailContactInfo * PostalContactInfo
```

Now let's say that the business decides that phone numbers need to be supported as well. The new business rule is: *"A contact must have at least one of the following: an email, a postal address, a home phone, or a work phone"*.

How can we represent this now?

A little thought reveals that there are 15 possible combinations of these four contact methods. Surely we don't want to create a union case with 15 choices? Is there a better way?

Let's hold that thought and look at a different but related problem.

Forcing breaking changes when requirements change

Here's the problem. Say that you have a contact structure which contains a list of email addresses and also a list of postal addresses, like so:

```
type ContactInformation =
    {
    EmailAddresses : EmailContactInfo list;
    PostalAddresses : PostalContactInfo list
    }
```

And, also let's say that you have created a `printReport` function that loops through the information and prints it out in a report:

```
// mock code
let printEmail emailAddress =
    printfn "Email Address is %s" emailAddress

// mock code
let printPostalAddress postalAddress =
    printfn "Postal Address is %s" postalAddress

let printReport contactInfo =
    let {
        EmailAddresses = emailAddresses;
        PostalAddresses = postalAddresses;
        } = contactInfo
    for email in emailAddresses do
         printEmail email
    for postalAddress in postalAddresses do
         printPostalAddress postalAddress
```

Crude, but simple and understandable.

Now if the new business rule comes into effect, we might decide to change the structure to have some new lists for the phone numbers. The updated structure will now look something like this:

```
type PhoneContactInfo = string // dummy for now

type ContactInformation =
    {
    EmailAddresses : EmailContactInfo list;
    PostalAddresses : PostalContactInfo list;
    HomePhones : PhoneContactInfo list;
    WorkPhones : PhoneContactInfo list;
    }
```

If you make this change, you also want to make sure that all the functions that process the contact infomation are updated to handle the new phone cases as well.

Certainly, you will be forced to fix any pattern matches that break. But in many cases, you would *not* be forced to handle the new cases.

For example, here's `printReport` updated to work with the new lists:

```
let printReport contactInfo =
    let {
        EmailAddresses = emailAddresses;
        PostalAddresses = postalAddresses;
        } = contactInfo
    for email in emailAddresses do
         printEmail email
    for postalAddress in postalAddresses do
         printPostalAddress postalAddress
```

Can you see the deliberate mistake? Yes, I forgot to change the function to handle the phones. The new fields in the record have not caused the code to break at all. There is no guarantee that you will remember to handle the new cases. It would be all too easy to forget.

Again, we have the challenge: can we design types such that these situations cannot easily happen?

Deeper insight into the domain

If you think about this example a bit more deeply, you will realize that we have missed the forest for the trees.

Our initial concept was: *"to contact a customer, there will be a list of possible emails, and a list of possible addresses, etc"*.

But really, this is all wrong. A much better concept is: *"To contact a customer, there will be a list of contact methods. Each contact method could be an email OR a postal address OR a phone number"*.

This is a key insight into how the domain should be modelled. It creates a whole new type, a "ContactMethod", which resolves our problems in one stroke.

We can immediately refactor the types to use this new concept:

```
type ContactMethod =
    | Email of EmailContactInfo
    | PostalAddress of PostalContactInfo
    | HomePhone of PhoneContactInfo
    | WorkPhone of PhoneContactInfo

type ContactInformation =
    {
    ContactMethods  : ContactMethod list;
    }
```

And the reporting code must now be changed to handle the new type as well:

```
// mock code
let printContactMethod cm =
    match cm with
    | Email emailAddress ->
        printfn "Email Address is %s" emailAddress
    | PostalAddress postalAddress ->
         printfn "Postal Address is %s" postalAddress
    | HomePhone phoneNumber ->
        printfn "Home Phone is %s" phoneNumber
    | WorkPhone phoneNumber ->
        printfn "Work Phone is %s" phoneNumber

let printReport contactInfo =
    let {
        ContactMethods=methods;
        } = contactInfo
    methods
    |> List.iter printContactMethod
```

These changes have a number of benefits.

First, from a modelling point of view, the new types represent the domain much better, and are more adaptable to changing requitements.

And from a development point of view, changing the type to be a union means that any new cases that we add (or remove) will break the code in a very obvious way, and it will be much harder to accidentally forget to handle all the cases.

Back to the business rule with 15 possible combinations

So now back to the original example. We left it thinking that, in order to encode the business rule, we might have to create 15 possible combinations of various contact methods.

But the new insight from the reporting problem also affects our understanding of the business rule.

With the "contact method" concept in our heads, we can rephase the requirement as: *"A customer must have at least one contact method. A contact method could be an email OR a postal addresses OR a phone number"*.

So let's redesign the `Contact` type to have a list of contact methods:

```
type Contact =
    {
    Name: PersonalName;
    ContactMethods: ContactMethod list;
    }
```

But this is still not quite right. The list could be empty. How can we enforce the rule that there must be *at least* one contact method?

The simplest way is to create a new field that is required, like this:

```
type Contact =
    {
    Name: PersonalName;
    PrimaryContactMethod: ContactMethod;
    SecondaryContactMethods: ContactMethod list;
    }
```

In this design, the `PrimaryContactMethod` is required, and the secondary contact methods are optional, which is exactly what the business rule requires!

And this refactoring too, has given us some insight. It may be that the concepts of "primary" and "secondary" contact methods might, in turn, clarify code in other areas, creating a cascading change of insight and refactoring.

Summary

In this post, we've seen how using types to model business rules can actually help you to understand the domain at a deeper level.

In the *Domain Driven Design* book, Eric Evans devotes a whole section and two chapters in particular (chapters 8 and 9) to discussing the importance of refactoring towards deeper insight. The example in this post is simple in comparison, but I hope that it shows that how an insight like this can help improve both the model and the code correctness.

In the next post, we'll see how types can help with representing fine-grained states.

Designing with types: Making state explicit

In this post we will look at making implicit states explicit by using state machines, and then modelling these state machines with union types.

Background

In an earlier post in this series, we looked at single case unions as a wrapper for types such as email addresses.

```
module EmailAddress =

    type T = EmailAddress of string

    let create (s:string) =
        if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
            then Some (EmailAddress s)
            else None
```

This code assumes that either an address is valid or it is not. If it is not, we reject it altogether and return `None` instead of a valid value.

But there are degrees of validity. For example, what happens if we want to keep an invalid email address around rather than just rejecting it? In this case, as usual, we want to use the type system to make sure that we don't get a valid address mixed up with an invalid address.

The obvious way to do this is with a union type:

```
module EmailAddress =

    type T =
        | ValidEmailAddress of string
        | InvalidEmailAddress of string

    let create (s:string) =
        if System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
            then ValidEmailAddress s    // change result type
            else InvalidEmailAddress s  // change result type

    // test
    let valid = create "abc@example.com"
    let invalid = create "example.com"
```

and with these types we can ensure that only valid emails get sent:

```
let sendMessageTo t =
    match t with
    | ValidEmailAddress email ->
        // send email
    | InvalidEmailAddress _ ->
        // ignore
```

So far, so good. This kind of design should be obvious to you by now.

But this approach is more widely applicable than you might think. In many situations, there are similar "states" that are not made explicit, and handled with flags, enums, or conditional logic in code.

State machines

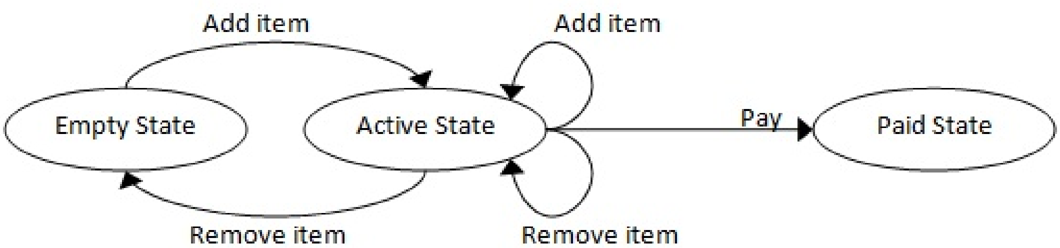
In the example above, the "valid" and "invalid" cases are mutually incompatible. That is, a valid email can never become invalid, and vice versa.

But in many cases, it is possible to go from one case to another, triggered by some kind of event. At which point we have a "state machine", where each case represents a "state", and moving from one state to another is a "transition".

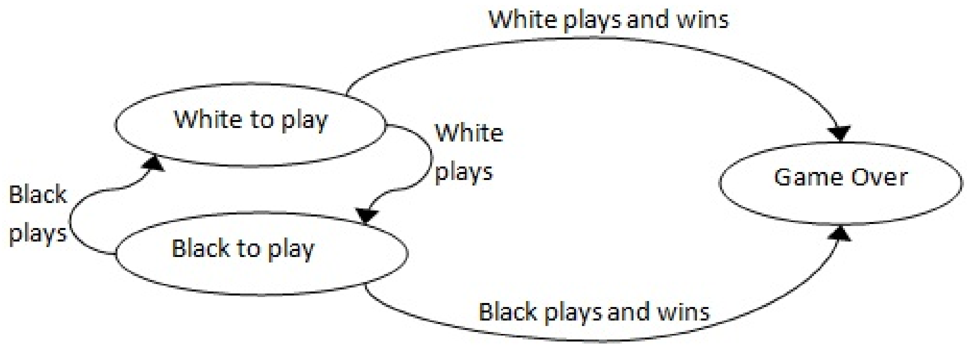
Some examples:

- A email address might have states "Unverified" and "Verified", where you can transition from the "Unverified" state to the "Verified" state by asking the user to click on a link in a 

confirmation email.
- A shopping cart might have states "Empty", "Active" and "Paid", where you can transition from the "Empty" state to the "Active" state by adding an item to the cart, and to the "Paid" state by paying.



- A game such as chess might have states "WhiteToPlay", "BlackToPlay" and "GameOver", where you can transition from the "WhiteToPlay" state to the "BlackToPlay" state by White making a non-game-ending move, or transition to the "GameOver" state by playing a checkmate move.



In each of these cases, we have a set of states, a set of transitions, and events that can trigger a transition.
State machines are often represented by a table, like this one for a shopping cart:

Current State	Event->	Add Item	Remove Item	Pay
Empty		new state = Active	n/a	n/a
Active		new state = Active	new state = Active or Empty, depending on the number of items	new state = Paid
Paid		n/a	n/a	n/a

With a table like this, you can quickly see exactly what should happen for each event when the system is in a given state.

Why use state machines?

There are a number of benefits to using state machines in these cases:

Each state can have different allowable behavior.

In the verified email example, there is probably a business rule that says that you can only send password resets to verified email addresses, not to unverified addresses. And in the shopping cart example, only an active cart can be paid for, and a paid cart cannot be added to.

All the states are explicitly documented.

It is all too easy to have important states that are implicit but never documented.

For example, the "empty cart" has different behavior from the "active cart" but it would be rare to see this documented explicitly in code.

It is a design tool that forces you to think about every possibility that could occur.

A common cause of errors is that certain edge cases are not handled, but a state machine forces all cases to be thought about.

For example, what should happen if we try to verify an already verified email? What happens if we try to remove an item from an empty shopping cart? What happens if white tries to play when the state is "BlackToPlay"? And so on.

How to implement simple state machines in F#

You are probably familiar with complex state machines, such as those used in language parsers and regular expressions. Those kinds of state machines are generated from rule sets or grammars, and are quite complicated.

The kinds of state machines that I'm talking about are much, much simpler. Just a few cases at the most, with a small number of transitions, so we don't need to use complex generators.

So what is the best way implement these simple state machines?

Typically, each state will have its own type, to store the data that is relevant to that state (if any), and then the entire set of states will be represented by a union class.

Here's an example using the shopping cart state machine:

```
type ActiveCartData = { UnpaidItems: string list }
type PaidCartData = { PaidItems: string list; Payment: float }

type ShoppingCart =
    | EmptyCart  // no data
    | ActiveCart of ActiveCartData
    | PaidCart of PaidCartData
```

Note that the `EmptyCart` state has no data, so no special type is needed.

Each event is then represented by a function that accepts the entire state machine (the union type) and returns a new version of the state machine (again, the union type).

Here's an example using two of the shopping cart events:

```
let addItem cart item =
    match cart with
    | EmptyCart ->
        // create a new active cart with one item
        ActiveCart {UnpaidItems=[item]}
    | ActiveCart {UnpaidItems=existingItems} ->
        // create a new ActiveCart with the item added
        ActiveCart {UnpaidItems = item :: existingItems}
    | PaidCart _ ->
        // ignore
        cart

let makePayment cart payment =
    match cart with
    | EmptyCart ->
        // ignore
        cart
    | ActiveCart {UnpaidItems=existingItems} ->
        // create a new PaidCart with the payment
        PaidCart {PaidItems = existingItems; Payment=payment}
    | PaidCart _ ->
        // ignore
        cart
```

You can see that from the caller's point of view, the set of states is treated as "one thing" for general manipulation (the `ShoppingCart` type), but when processing the events internally, each state is treated separately.

Designing event handling functions

Guideline: *Event handling functions should always accept and return the entire state machine*

You might ask: why do we have to pass in the whole shopping cart to the event-handling functions? For example, the `makePayment` event only has relevance when the cart is in the Active state, so why not just explicitly pass it the ActiveCart type, like this:

```
let makePayment2 activeCart payment =
    let {UnpaidItems=existingItems} = activeCart
    {PaidItems = existingItems; Payment=payment}
```

Let's compare the function signatures:

```
// the original function
val makePayment : ShoppingCart -> float -> ShoppingCart

// the new more specific function
val makePayment2 :  ActiveCartData -> float -> PaidCartData
```

You will see that the original `makePayment` function takes a cart and results in a cart, while the new function takes an `ActiveCartData` and results in a `PaidCartData`, which seems to be much more relevant.

But if you did this, how would you handle the same event when the cart was in a different state, such as empty or paid? Someone has to handle the event for all three possible states somewhere, and it is much better to encapsulate this business logic inside the function than to be at the mercy of the caller.

Working with "raw" states

Occasionally you do genuinely need to treat one of the states as a separate entity in its own right and use it independently. Because each state is a type as well, this is normally straightforward.

For example, if I need to report on all paid carts, I can pass it a list of `PaidCartData`.

```
let paymentReport paidCarts =
    let printOneLine {Payment=payment} =
        printfn "Paid %f for items" payment
    paidCarts |> List.iter printOneLine
```

By using a list of `PaidCartData` as the parameter rather than `ShoppingCart` itself, I ensure that I cannot accidentally report on unpaid carts.

If you do this, it should be in a supporting function to the event handlers, never the event handlers themselves.

Using explicit states to replace boolean flags

Let's look at how we can apply this approach to a real example now.

In the `Contact` example from an earlier post we had a flag that was used to indicate whether a customer had verified their email address. The type looked like this:

```
type EmailContactInfo =
    {
    EmailAddress: EmailAddress.T;
    IsEmailVerified: bool;
    }
```

Any time you see a flag like this, chances are you are dealing with state. In this case, the boolean is used to indicate that we have two states: "Unverified" and "Verified".

As mentioned above, there will probably be various business rules associated with what is permissible in each state. For example, here are two:

- Business rule: *"Verification emails should only be sent to customers who have unverified email addresses"*
- Business rule: *"Password reset emails should only be sent to customers who have verified email addresses"*

As before, we can use types to ensure that code conforms to these rules.

Let's rewrite the `EmailContactInfo` type using a state machine. We'll put it in an module as well.

We'll start by defining the two states.

- For the "Unverified" state, the only data we need to keep is the email address.
- For the "Verified" state, we might want to keep some extra data in addition to the email address, such as the date it was verified, the number of recent password resets, on so on. This data is not relevant (and should not even be visible) to the "Unverified" state.

```
module EmailContactInfo =
    open System

    // placeholder
    type EmailAddress = string

    // UnverifiedData = just the email
    type UnverifiedData = EmailAddress

    // VerifiedData = email plus the time it was verified
    type VerifiedData = EmailAddress * DateTime

    // set of states
    type T =
        | UnverifiedState of UnverifiedData
        | VerifiedState of VerifiedData
```

Note that for the `UnverifiedData` type I just used a type alias. No need for anything more complicated right now, but using a type alias makes the purpose explicit and helps with refactoring.

Now let's handle the construction of a new state machine, and then the events.

- Construction *always* results in an unverified email, so that is easy.
- There is only one event that transitions from one state to another: the "verified" event.

```
module EmailContactInfo =

    // types as above

    let create email =
        // unverified on creation
        UnverifiedState email

    // handle the "verified" event
    let verified emailContactInfo dateVerified =
        match emailContactInfo with
        | UnverifiedState email ->
            // construct a new info in the verified state
            VerifiedState (email, dateVerified)
        | VerifiedState _ ->
            // ignore
            emailContactInfo
```

Note that, as discussed here, every branch of the match must return the same type, so when ignoring the verified state we must still return something, such as the object that was passed in.

Finally, we can write the two utility functions `sendVerificationEmail` and `sendPasswordReset`.

```
module EmailContactInfo =

    // types and functions as above

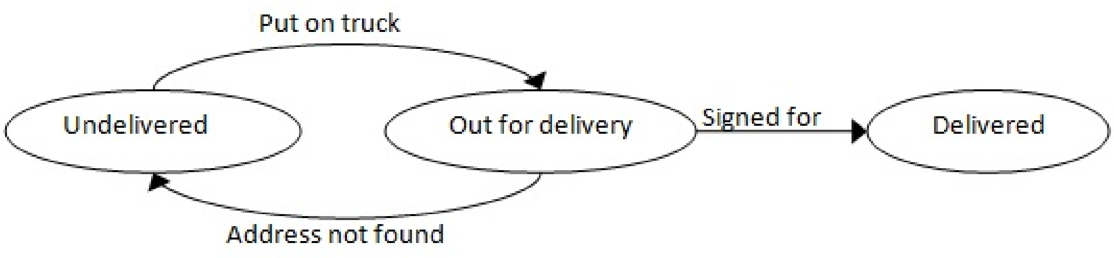
    let sendVerificationEmail emailContactInfo =
        match emailContactInfo with
        | UnverifiedState email ->
            // send email
            printfn "sending email"
        | VerifiedState _ ->
            // do nothing
            ()

    let sendPasswordReset emailContactInfo =
        match emailContactInfo with
        | UnverifiedState email ->
            // ignore
            ()
        | VerifiedState _ ->
            // ignore
            printfn "sending password reset"
```

Using explicit cases to replace case/switch statements

Sometimes it is not just a simple boolean flag that is used to indicate state. In C# and Java it is common to use a `int` or an `enum` to represent a set of states.

For example, here's a simple state diagram of a package status for a delivery system, where a package has three possible states:



There are some obvious business rules that come out of this diagram:

- *Rule: "You can't put a package on a truck if it is already out for delivery"*
- *Rule: "You can't sign for a package that is already delivered"*

and so on.

Now, without using union types, we might represent this design by using an enum to represent the state, like this:

```
open System

type PackageStatus =
    | Undelivered
    | OutForDelivery
    | Delivered

type Package =
    {
    PackageId: int;
    PackageStatus: PackageStatus;
    DeliveryDate: DateTime;
    DeliverySignature: string;
    }
```

And then the code to handle the "putOnTruck" and "signedFor" events might look like this:

```
let putOnTruck package =
    {package with PackageStatus=OutForDelivery}

let signedFor package signature =
    let {PackageStatus=packageStatus} = package
    if (packageStatus = Undelivered)
    then
        failwith "package not out for delivery"
    else if (packageStatus = OutForDelivery)
    then
        {package with
            PackageStatus=OutForDelivery;
            DeliveryDate = DateTime.UtcNow;
            DeliverySignature=signature;
            }
    else
        failwith "package already delivered"
```

This code has some subtle bugs in it.

- When handling the "putOnTruck" event, what should happen in the case that the status is *already* `OutForDelivery` or `Delivered`. The code is not explicit about it.
- When handling the "signedFor" event, we do handle the other states, but the last else branch assumes that we only have three states, and therefore doesn't bother to be explicit about testing for it. This code would be incorrect if we ever added a new status.
- Finally, because the `DeliveryDate` and `DeliverySignature` are in the basic structure, it would be possible to set them accidentally, even though the status was not `Delivered`.

But as usual, the idiomatic and more type-safe F# approach is to use an overall union type rather than embed a status value inside a data structure.

```
open System

type UndeliveredData =
    {
    PackageId: int;
    }

type OutForDeliveryData =
    {
    PackageId: int;
    }

type DeliveredData =
    {
    PackageId: int;
    DeliveryDate: DateTime;
    DeliverySignature: string;
    }

type Package =
    | Undelivered of UndeliveredData
    | OutForDelivery of OutForDeliveryData
    | Delivered of DeliveredData
```

And then the event handlers *must* handle every case.

```
let putOnTruck package =
    match package with
    | Undelivered {PackageId=id} ->
        OutForDelivery {PackageId=id}
    | OutForDelivery _ ->
        failwith "package already out"
    | Delivered _ ->
        failwith "package already delivered"

let signedFor package signature =
    match package with
    | Undelivered _ ->
        failwith "package not out"
    | OutForDelivery {PackageId=id} ->
        Delivered {
            PackageId=id;
            DeliveryDate = DateTime.UtcNow;
            DeliverySignature=signature;
            }
    | Delivered _ ->
        failwith "package already delivered"
```

Note: I am using `failWith` to handle the errors. In a production system, this code should be replaced by client driven error handlers. See the discussion of handling constructor errors in the post about single case DUs for some ideas.

Using explicit cases to replace implicit conditional code

Finally, there are often cases where a system has states, but they are implicit in conditional code.

For example, here is a type that represents an order.

```
open System

type Order =
    {
    OrderId: int;
    PlacedDate: DateTime;
    PaidDate: DateTime option;
    PaidAmount: float option;
    ShippedDate: DateTime option;
    ShippingMethod: string option;
    ReturnedDate: DateTime option;
    ReturnedReason: string option;
    }
```

You can guess that Orders can be "new", "paid", "shipped" or "returned", and have timestamps and extra information for each transition, but this is not made explicit in the structure.

The option types are a clue that this type is trying to do too much. At least F# forces you to use options -- in C# or Java these might just be nulls, and you would have no idea from the type definition whether they were required or not.

And now let's look at the kind of ugly code that might test these option types to see what state the order is in.

Again, there is some important business logic that depends on the state of the order, but nowhere is it explicitly documented what the various states and transitions are.

```
let makePayment order payment =
    if (order.PaidDate.IsSome)
    then failwith "order is already paid"
    //return an updated order with payment info
    {order with
        PaidDate=Some DateTime.UtcNow
        PaidAmount=Some payment
        }

let shipOrder order shippingMethod =
    if (order.ShippedDate.IsSome)
    then failwith "order is already shipped"
    //return an updated order with shipping info
    {order with
        ShippedDate=Some DateTime.UtcNow
        ShippingMethod=Some shippingMethod
        }
```

Note: I added `IsSome` to test for option values being present as a direct port of the way that a C# program would test for `null`. But `IsSome` is both ugly and dangerous. Don't use it!

Here is a better approach using types that makes the states explicit.

```
open System

type InitialOrderData =
    {
    OrderId: int;
    PlacedDate: DateTime;
    }
type PaidOrderData =
    {
    Date: DateTime;
    Amount: float;
    }
type ShippedOrderData =
    {
    Date: DateTime;
    Method: string;
    }
type ReturnedOrderData =
    {
    Date: DateTime;
    Reason: string;
    }

type Order =
    | Unpaid of InitialOrderData
    | Paid of InitialOrderData * PaidOrderData
    | Shipped of InitialOrderData * PaidOrderData * ShippedOrderData
    | Returned of InitialOrderData * PaidOrderData * ShippedOrderData * ReturnedOrderD
ata
```

And here are the event handling methods:

```
let makePayment order payment =
    match order with
    | Unpaid i ->
        let p = {Date=DateTime.UtcNow; Amount=payment}
        // return the Paid order
        Paid (i,p)
    | _ ->
        printfn "order is already paid"
        order

let shipOrder order shippingMethod =
    match order with
    | Paid (i,p) ->
        let s = {Date=DateTime.UtcNow; Method=shippingMethod}
        // return the Shipped order
        Shipped (i,p,s)
    | Unpaid _ ->
        printfn "order is not paid for"
        order
    | _ ->
        printfn "order is already shipped"
        order
```

Note: Here I am using `printfn` to handle the errors. In a production system, do use a different approach.

When not to use this approach

As with any technique we learn, we have to be careful of treating it like a golden hammer.

This approach does add complexity, so before you start using it, be sure that benefits will outweigh the costs.

To recap, here are the conditions where using simple state machines might be benficial:

- You have a set of mutually exclusive states with transitions between them.
- The transitions are triggered by external events.
- The states are exhaustive. That is, there are no other choices and you must always handle all cases.
- Each state might have associated data that should not be accessable when the system is in another state.
- There are static business rules that apply to the states.

Let's look at some examples where these guidelines *don't* apply.

States are not important in the domain.

Consider a blog authoring application. Typically, each blog post can be in a state such as "Draft", "Published", etc. And there are obviously transitions between these states driven by events (such as clicking a "publish" button).

But is it worth creating a state machine for this? Generally, I would say not.

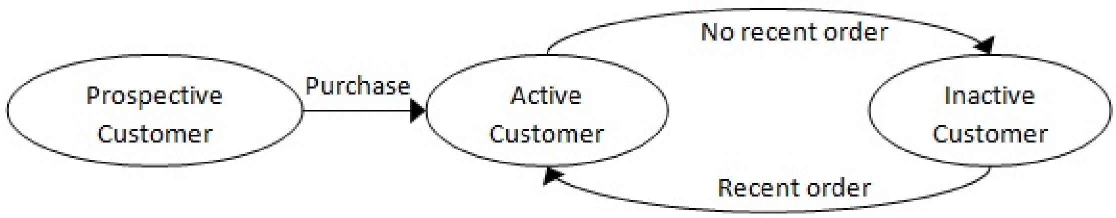
Yes, there are state transitions, but is there really any change in logic because of this? From the authoring point of view, most blogging apps don't have any restrictions based on the state. You can author a draft post in exactly the same way as you author a published post.

The only part of the system that *does* care about the state is the display engine, and that filters out the drafts in the database layer before it ever gets to the domain.

Since there is no special domain logic that cares about the state, it is probably unnecessary.

State transitions occur outside the application

In a customer management application, it is common to classify customers as "prospects", "active", "inactive", etc.



In the application, these states have business meaning and should be represented by the type system (such as a union type). But the state *transitions* generally do not occur within the application itself. For example, we might classify a customer as inactive if they haven't ordered anything for 6 months. And then this rule might be applied to customer records in a database by a nightly batch job, or when the customer record is loaded from the database. But from our application's point of view, the transitions do not happen *within* the application, and so we do not need to create a special state machine.

Dynamic business rules

The last bullet point in the list above refers to "static" business rules. By this I mean that the rules change slowly enough that they should be embedded into the code itself.

On the other hand, if the rules are dynamic and change frequently, it is probably not worth going to the trouble of creating static types.

In these situations, you should consider using active patterns, or even a proper rules engine.

Summary

In this post, we've seen that if you have data structures with explicit flags ("IsVerified") or status fields ("OrderStatus"), or implicit state (clued by an excessive number of nullable or option types), it is worth considering using a simple state machine to model the domain objects. In most cases the extra complexity is compensated for by the explicit documention of the states and the elimination of errors due to not handling all possible cases.

Designing with types: Constrained strings

In a previous post, I talked about avoiding using plain primitive strings for email addresses, zip codes, states, etc. By wrapping them in a single case union, we could (a) force the types to be distinct and (b) add validation rules.

In this post, we'll look at whether we can extend that concept to an even more fine grained level.

When is a string not a string?

Let's look a simple `PersonalName` type.

```
type PersonalName =
    {
    FirstName: string;
    LastName: string;
    }
```

The type says that the first name is a `string`. But really, is that all it is? Are there any other constraints that we might need to add to it?

Well, OK, it must not be null. But that is assumed in F#.

What about the length of the string? Is it acceptable to have a name which is 64K characters long? If not, then is there some maximum length allowed?

And can a name contain linefeed characters or tabs? Can it start or end with whitespace?

Once you put it this way, there are quite a lot of constraints even for a "generic" string. Here are some of the obvious ones:

- What is its maximum length?
- Can it cross over multiple lines?
- Can it have leading or trailing whitespace?
- Can it contain non-printing characters?

Should these constraints be part of the domain model?

So we might acknowledge that some constraints exist, but should they really be part of the domain model (and the corresponding types derived from it)? For example, the constraint that a last name is limited to 100 characters -- surely that is specific to a particular implementation and not part of the domain at all.

I would answer that there is a difference between a logical model and a physical model. In a logical model some of these constraints might not be relevant, but in a physical model they most certainly are. And when we are writing code, we are always dealing with a physical model anyway.

Another reason for incorporating the constraints into the model is that often the model is shared across many separate applications. For example, a personal name may be created in a e-commerce application, which writes it into a database table and then puts it on a message queue to be picked up by a CRM application, which in turn calls an email templating service, and so on.

It is important that all these applications and services have the *same* idea of what a personal name is, including the length and other constraints. If the model does not make the constraints explicit, then it is easy to have a mismatch when moving across service boundaries.

For example, have you ever written code that checks the length of a string before writing it to a database?

```
void SaveToDatabase(PersonalName personalName)
{
   var first = personalName.First;
   if (first.Length > 50)
   {
        // ensure string is not too long
        first = first.Substring(0,50);
   }

   //save to database
}
```

If the string *is* too long at this point, what should you do? Silently truncate it? Throw an exception?

A better answer is to avoid the problem altogether if you can. By the time the string gets to the database layer it is too late -- the database layer should not be making these kinds of decisions.

The problem should be dealt with when the string was *first created*, not when it is *used*. In other words, it should have been part of the validation of the string.

But how can we trust that the validation has been done correctly for all possible paths? I think you can guess the answer...

Modeling constrained strings with types

The answer, of course, is to create wrapper types which have the constraints built into the type.

So let's knock up a quick prototype using the single case union technique we used before.

```
module String100 =
    type T = String100 of string
    let create (s:string) =
        if s <> null && s.Length <= 100
        then Some (String100 s)
        else None
    let apply f (String100 s) = f s
    let value s = apply id s

module String50 =
    type T = String50 of string
    let create (s:string) =
        if s <> null && s.Length <= 50
        then Some (String50 s)
        else None
    let apply f (String50 s) = f s
    let value s = apply id s

module String2 =
    type T = String2 of string
    let create (s:string) =
        if s <> null && s.Length <= 2
        then Some (String2 s)
        else None
    let apply f (String2 s) = f s
    let value s = apply id s
```

Note that we immediately have to deal with the case when the validation fails by using an option type as the result. It makes creation more painful, but we can't avoid it if we want the benefits later.

For example, here is a good string and a bad string of length 2.

```
let s2good = String2.create "CA"
let s2bad = String2.create "California"

match s2bad with
| Some s2 -> // update domain object
| None -> // handle error
```

In order to use the `String2` value we are forced to check whether it is `Some` or `None` at the time of creation.

Problems with this design

One problem is that we have a lot of duplicated code. In practice a typical domain only has a few dozen string types, so there won't be that much wasted code. But still, we can probably do better.

Another more serious problem is that comparisons become harder. A `String50` is a different type from a `String100` so that they cannot be compared directly.

```
let s50 = String50.create "John"
let s100 = String100.create "Smith"

let s50' = s50.Value
let s100' = s100.Value

let areEqual = (s50' = s100')  // compiler error
```

This kind of thing will make working with dictionaries and lists harder.

Refactoring

At this point we can exploit F#'s support for interfaces, and create a common interface that all wrapped strings have to support, and also some standard functions:

```
module WrappedString =

    /// An interface that all wrapped strings support
    type IWrappedString =
        abstract Value : string

    /// Create a wrapped value option
    /// 1) canonicalize the input first
    /// 2) If the validation succeeds, return Some of the given constructor
    /// 3) If the validation fails, return None
    /// Null values are never valid.
    let create canonicalize isValid ctor (s:string) =
        if s = null
        then None
        else
            let s' = canonicalize s
            if isValid s'
            then Some (ctor s')
            else None

    /// Apply the given function to the wrapped value
    let apply f (s:IWrappedString) =
        s.Value |> f

    /// Get the wrapped value
    let value s = apply id s

    /// Equality test
    let equals left right =
        (value left) = (value right)

    /// Comparison
    let compareTo left right =
        (value left).CompareTo (value right)
```

The key function is `create`, which takes a constructor function and creates new values using it only when the validation passes.

With this in place it is a lot easier to define new types:

```
module WrappedString =

    // ... code from above ...

    /// Canonicalizes a string before construction
    /// * converts all whitespace to a space char
    /// * trims both ends
    let singleLineTrimmed s =
        System.Text.RegularExpressions.Regex.Replace(s,"\s"," ").Trim()

    /// A validation function based on length
    let lengthValidator len (s:string) =
        s.Length <= len

    /// A string of length 100
    type String100 = String100 of string with
        interface IWrappedString with
            member this.Value = let (String100 s) = this in s

    /// A constructor for strings of length 100
    let string100 = create singleLineTrimmed (lengthValidator 100) String100

    /// Converts a wrapped string to a string of length 100
    let convertTo100 s = apply string100 s

    /// A string of length 50
    type String50 = String50 of string with
        interface IWrappedString with
            member this.Value = let (String50 s) = this in s

    /// A constructor for strings of length 50
    let string50 = create singleLineTrimmed (lengthValidator 50)  String50

    /// Converts a wrapped string to a string of length 50
    let convertTo50 s = apply string50 s
```

For each type of string now, we just have to:

- create a type (e.g. `String100`)
- an implementation of `IWrappedString` for that type
- and a public constructor (e.g. `string100`) for that type.

(In the sample above I have also thrown in a useful `convertTo` to convert from one type to another.)

The type is a simple wrapped type as we have seen before.

The implementation of the `Value` method of the IWrappedString could have been written using multiple lines, like this:

```
member this.Value =
    let (String100 s) = this
    s
```

But I chose to use a one liner shortcut:

```
member this.Value = let (String100 s) = this in s
```

The constructor function is also very simple. The canonicalize function is `singleLineTrimmed` , the validator function checks the length, and the constructor is the `String100` function (the function associated with the single case, not to be confused with the type of the same name).

```
let string100 = create singleLineTrimmed (lengthValidator 100) String100
```

If you want to have other types with different constraints, you can easily add them. For example you might want to have a `Text1000` type that supports multiple lines and embedded tabs and is not trimmed.

```
module WrappedString =

    // ... code from above ...

    /// A multiline text of length 1000
    type Text1000 = Text1000 of string with
        interface IWrappedString with
            member this.Value = let (Text1000 s) = this in s

    /// A constructor for multiline strings of length 1000
    let text1000 = create id (lengthValidator 1000) Text1000
```

Playing with the WrappedString module

We can now play with the module interactively to see how it works:

```
let s50 = WrappedString.string50 "abc" |> Option.get
printfn "s50 is %A" s50
let bad = WrappedString.string50 null
printfn "bad is %A" bad
let s100 = WrappedString.string100 "abc" |> Option.get
printfn "s100 is %A" s100

// equality using module function is true
printfn "s50 is equal to s100 using module equals? %b" (WrappedString.equals s50 s100)

// equality using Object method is false
printfn "s50 is equal to s100 using Object.Equals? %b" (s50.Equals s100)

// direct equality does not compile
printfn "s50 is equal to s100? %b" (s50 = s100) // compiler error
```

When we need to interact with types such as maps that use raw strings, it is easy to compose new helper functions.

For example, here are some helpers to work with maps:

```
module WrappedString =

    // ... code from above ...

    /// map helpers
    let mapAdd k v map =
        Map.add (value k) v map

    let mapContainsKey k map =
        Map.containsKey (value k) map

    let mapTryFind k map =
        Map.tryFind (value k) map
```

And here is how these helpers might be used in practice:

```
let abc = WrappedString.string50 "abc" |> Option.get
let def = WrappedString.string100 "def" |> Option.get
let map =
    Map.empty
    |> WrappedString.mapAdd abc "value for abc"
    |> WrappedString.mapAdd def "value for def"

printfn "Found abc in map? %A" (WrappedString.mapTryFind abc map)

let xyz = WrappedString.string100 "xyz" |> Option.get
printfn "Found xyz in map? %A" (WrappedString.mapTryFind xyz map)
```

So overall, this "WrappedString" module allows us to create nicely typed strings without interfering too much. Now let's use it in a real situation.

Using the new string types in the domain

Now we have our types, we can change the definition of the `PersonalName` type to use them.

```
module PersonalName =
    open WrappedString

    type T =
        {
        FirstName: String50;
        LastName: String100;
        }

    /// create a new value
    let create first last =
        match (string50 first),(string100 last) with
        | Some f, Some l ->
            Some {
                FirstName = f;
                LastName = l;
                }
        | _ ->
            None
```

We have created a module for the type and added a creation function that converts a pair of strings into a `PersonalName`.

Note that we have to decide what to do if *either* of the input strings are invalid. Again, we cannot postpone the issue till later, we have to deal with it at construction time.

In this case we use the simple approach of creating an option type with None to indicate failure.

Here it is in use:

```
let name = PersonalName.create "John" "Smith"
```

We can also provide additional helper functions in the module.

Let's say, for example, that we want to create a `fullname` function that will return the first and last names joined together.

Again, more decisions to make.

- Should we return a raw string or a wrapped string? The advantage of the latter is that the callers know exactly how long the string will be, and it will be compatible with other similar types.
- If we do return a wrapped string (say a `String100`), then how do we handle the the case when the combined length is too long? (It could be up to 151 chars, based on the length of the first and last name types.). We could either return an option, or force a truncation if the combined length is too long.

Here's code that demonstrates all three options.

```
module PersonalName =

    // ... code from above ...

    /// concat the first and last names together
    /// and return a raw string
    let fullNameRaw personalName =
        let f = personalName.FirstName |> value
        let l = personalName.LastName |> value
        f + " " + l

    /// concat the first and last names together
    /// and return None if too long
    let fullNameOption personalName =
        personalName |> fullNameRaw |> string100

    /// concat the first and last names together
    /// and truncate if too long
    let fullNameTruncated personalName =
        // helper function
        let left n (s:string) =
            if (s.Length > n)
            then s.Substring(0,n)
            else s

        personalName
        |> fullNameRaw  // concat
        |> left 100     // truncate
        |> string100    // wrap
        |> Option.get   // this will always be ok
```

Which particular approach you take to implementing `fullName` is up to you. But it demonstrates a key point about this style of type-oriented design: these decisions have to be taken *up front*, when creating the code. You cannot postpone them till later.

This can be very annoying at times, but overall I think it is a good thing.

Revisiting the email address and zip code types

We can use this WrappedString module to reimplement the `EmailAddress` and `ZipCode` types.

```
module EmailAddress =

    type T = EmailAddress of string with
        interface WrappedString.IWrappedString with
            member this.Value = let (EmailAddress s) = this in s

    let create =
        let canonicalize = WrappedString.singleLineTrimmed
        let isValid s =
            (WrappedString.lengthValidator 100 s) &&
            System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        WrappedString.create canonicalize isValid EmailAddress

    /// Converts any wrapped string to an EmailAddress
    let convert s = WrappedString.apply create s

module ZipCode =

    type T = ZipCode of string with
        interface WrappedString.IWrappedString with
            member this.Value = let (ZipCode s) = this in s

    let create =
        let canonicalize = WrappedString.singleLineTrimmed
        let isValid s =
            System.Text.RegularExpressions.Regex.IsMatch(s,@"^\d{5}$")
        WrappedString.create canonicalize isValid ZipCode

    /// Converts any wrapped string to a ZipCode
    let convert s = WrappedString.apply create s
```

Other uses of wrapped strings

This approach to wrapping strings can also be used for other scenarios where you don't want to mix string types together accidentally.

One case that leaps to mind is ensuring safe quoting and unquoting of strings in web applications.

For example, let's say that you want to output a string to HTML. Should the string be escaped or not?
If it is already escaped, you want to leave it alone but if it is not, you do want to escape it.

This can be a tricky problem. Joel Spolsky discusses using a naming convention here, but of course, in F#, we want a type-based solution instead.

A type-based solution will probably use a type for "safe" (already escaped) HTML strings (`HtmlString` say), and one for safe Javascript strings (`JsString`), one for safe SQL strings (`SqlString`), etc. Then these strings can be mixed and matched safely without accidentally causing security issues.

I won't create a solution here (and you will probably be using something like Razor anyway), but if you are interested you can read about a Haskell approach here and a port of that to F#.

Update

Many people have asked for more information on how to ensure that constrained types such as `EmailAddress` are only created through a special constructor that does the validation. So I have created a gist here that has some detailed examples of other ways of doing it.

Designing with types: Non-string types

In this series we've seen a lot of uses of single case discriminated unions to wrap strings.

There is no reason why you cannot use this technique with other primitive types, such as numbers and dates. Let's look a few examples.

Single case unions

In many cases, we want to avoid accidentally mixing up different kinds of integers. Two domain objects may have the same representation (using integers) but they should never be confused.

For example, you may have an `OrderId` and a `CustomerId`, both of which are stored as ints. But they are not *really* ints. You cannot add 42 to a `CustomerId`, for example. And `CustomerId(42)` is not equal to `OrderId(42)`. In fact, they should not even be allowed to be compared at all.

Types to the rescue, of course.

```
type CustomerId = CustomerId of int
type OrderId = OrderId of int

let custId = CustomerId 42
let orderId = OrderId 42

// compiler error
printfn "cust is equal to order? %b" (custId = orderId)
```

Similarly, you might want avoid mixing up semantically different date values by wrapping them in a type. (`DateTimeKind` is an attempt at this, but not always reliable.)

```
type LocalDttm = LocalDttm of System.DateTime
type UtcDttm = UtcDttm of System.DateTime
```

With these types we can ensure that we always pass the right kind of datetime as parameters. Plus, it acts as documentation as well.

```
let SetOrderDate (d:LocalDttm) =
    () // do something

let SetAuditTimestamp (d:UtcDttm) =
    () // do something
```

Constraints on integers

Just as we had validation and constraints on types such as `String50` and `ZipCode`, we can use the same approach when we need to have constraints on integers.

For example, an inventory management system or a shopping cart may require that certain types of number are always positive. You might ensure this by creating a `NonNegativeInt` type.

```
module NonNegativeInt =
    type T = NonNegativeInt of int

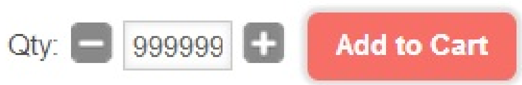
    let create i =
        if (i >= 0 )
        then Some (NonNegativeInt i)
        else None

module InventoryManager =

    // example of NonNegativeInt in use
    let SetStockQuantity (i:NonNegativeInt.T) =
        //set stock
        ()
```

Embedding business rules in the type

Just as we wondered earlier whether first names could ever be 64K characters long, can you really add 999999 items to your shopping cart?



Is it worth trying to avoid this issue by using constrained types? Let's look at some real code.

Here is a very simple shopping cart manager using a standard `int` type for the quantity. The quantity is incremented or decremented when the related buttons are clicked. Can you find the obvious bug?

```
module ShoppingCartWithBug =

    let mutable itemQty = 1  // don't do this at home!

    let incrementClicked() =
        itemQty <- itemQty + 1

    let decrementClicked() =
        itemQty <- itemQty - 1
```

If you can't quickly find the bug, perhaps you should consider making any constraints more explicit.

Here is the same simple shopping cart manager using a typed quantity instead. Can you find the bug now? (Tip: paste the code into a F# script file and run it)

```
module ShoppingCartQty =

    type T = ShoppingCartQty of int

    let initialValue = ShoppingCartQty 1

    let create i =
        if (i > 0 && i < 100)
        then Some (ShoppingCartQty i)
        else None

    let increment t = create (t + 1)
    let decrement t = create (t - 1)

module ShoppingCartWithTypedQty =

    let mutable itemQty = ShoppingCartQty.initialValue

    let incrementClicked() =
        itemQty <- ShoppingCartQty.increment itemQty

    let decrementClicked() =
        itemQty <- ShoppingCartQty.decrement itemQty
```

You might think this is overkill for such a trivial problem. But if you want to avoid being in the DailyWTF, it might be worth considering.

Constraints on dates

Not all systems can handle all possible dates. Some systems can only store dates going back to 1/1/1980, and some systems can only go into the future up to 2038 (I like to use 1/1/2038 as a max date to avoid US/UK issues with month/day order).

As with integers, it might be useful to have constraints on the valid dates built into the type, so that any out of bound issues are dealt with at construction time rather than later on.

```
type SafeDate = SafeDate of System.DateTime

let create dttm =
    let min = new System.DateTime(1980,1,1)
    let max = new System.DateTime(2038,1,1)
    if dttm < min || dttm > max
    then None
    else Some (SafeDate dttm)
```

Union types vs. units of measure

You might be asking at this point: What about units of measure? Aren't they meant to be used for this purpose?

Yes and no. Units of measure can indeed be used to avoid mixing up numeric values of different type, and are much more powerful than the single case unions we've been using.

On the other hand, units of measure are not encapsulated and cannot have constraints. Anyone can create a int with unit of measure `<kg>` say, and there is no min or max value.

In many cases, both approaches will work fine. For example, there are many parts of the .NET library that use timeouts, but sometimes the timeouts are set in seconds, and sometimes in milliseconds. I often have trouble remembering which is which. I definitely don't want to accidentally use a 1000 second timeout when I really meant a 1000 millisecond timeout.

To avoid this scenario, I often like to create separate types for seconds and milliseconds.

Here's a type based approach using single case unions:

```
type TimeoutSecs = TimeoutSecs of int
type TimeoutMs = TimeoutMs of int

let toMs (TimeoutSecs secs)  =
    TimeoutMs (secs * 1000)

let toSecs (TimeoutMs ms) =
    TimeoutSecs (ms / 1000)

/// sleep for a certain number of milliseconds
let sleep (TimeoutMs ms) =
    System.Threading.Thread.Sleep ms

/// timeout after a certain number of seconds
let commandTimeout (TimeoutSecs s) (cmd:System.Data.IDbCommand) =
    cmd.CommandTimeout <- s
```

And here's the same thing using units of measure:

```
[<Measure>] type sec
[<Measure>] type ms

let toMs (secs:int<sec>) =
    secs * 1000<ms/sec>

let toSecs (ms:int<ms>) =
    ms / 1000<ms/sec>

/// sleep for a certain number of milliseconds
let sleep (ms:int<ms>) =
    System.Threading.Thread.Sleep (ms * 1<_>)

/// timeout after a certain number of seconds
let commandTimeout (s:int<sec>) (cmd:System.Data.IDbCommand) =
    cmd.CommandTimeout <- (s * 1<_>)
```

Which approach is better?

If you are doing lots of arithmetic on them (adding, multiplying, etc) then the units of measure approach is much more convenient, but otherwise there is not much to choose between them.

Designing with types: Conclusion

In this series, we've looked at some of the ways we can use types as part of the design process, including:

- Breaking large structures down into small "atomic" components.
- Using single case unions to add semantic meaning and validation to key domain types such `EmailAddress` and `ZipCode`.
- Ensuring that the type system can only represent valid data ("making illegal states unrepresentable").
- Using types as an analysis tool to uncover hidden requirements
- Replacing flags and enums with simple state machines
- Replacing primitive strings with types that guarantee various constraints

For this final post, let's see them all applied together.

The "before" code

Here's the original example we started off with in the first post in the series:

```
type Contact =
    {
    FirstName: string;
    MiddleInitial: string;
    LastName: string;

    EmailAddress: string;
    //true if ownership of email address is confirmed
    IsEmailVerified: bool;

    Address1: string;
    Address2: string;
    City: string;
    State: string;
    Zip: string;
    //true if validated against address service
    IsAddressValid: bool;
    }
```

And how does that compare to the final result after applying all the techniques above?

The "after" code

First, let's start with the types that are not application specific. These types could probably be reused in many applications.

```
// ========================================
// WrappedString
// ========================================

/// Common code for wrapped strings
module WrappedString =

    /// An interface that all wrapped strings support
    type IWrappedString =
        abstract Value : string

    /// Create a wrapped value option
    /// 1) canonicalize the input first
    /// 2) If the validation succeeds, return Some of the given constructor
    /// 3) If the validation fails, return None
    /// Null values are never valid.
    let create canonicalize isValid ctor (s:string) =
        if s = null
        then None
        else
            let s' = canonicalize s
            if isValid s'
            then Some (ctor s')
            else None

    /// Apply the given function to the wrapped value
    let apply f (s:IWrappedString) =
        s.Value |> f

    /// Get the wrapped value
    let value s = apply id s

    /// Equality
    let equals left right =
        (value left) = (value right)

    /// Comparison
    let compareTo left right =
        (value left).CompareTo (value right)

    /// Canonicalizes a string before construction
    /// * converts all whitespace to a space char
    /// * trims both ends
    let singleLineTrimmed s =
        System.Text.RegularExpressions.Regex.Replace(s,"\s"," ").Trim()
```

```
    /// A validation function based on length
    let lengthValidator len (s:string) =
        s.Length <= len

    /// A string of length 100
    type String100 = String100 of string with
        interface IWrappedString with
            member this.Value = let (String100 s) = this in s

    /// A constructor for strings of length 100
    let string100 = create singleLineTrimmed (lengthValidator 100) String100

    /// Converts a wrapped string to a string of length 100
    let convertTo100 s = apply string100 s

    /// A string of length 50
    type String50 = String50 of string with
        interface IWrappedString with
            member this.Value = let (String50 s) = this in s

    /// A constructor for strings of length 50
    let string50 = create singleLineTrimmed (lengthValidator 50)  String50

    /// Converts a wrapped string to a string of length 50
    let convertTo50 s = apply string50 s

    /// map helpers
    let mapAdd k v map =
        Map.add (value k) v map

    let mapContainsKey k map =
        Map.containsKey (value k) map

    let mapTryFind k map =
        Map.tryFind (value k) map

// ========================================
// Email address (not application specific)
// ========================================

module EmailAddress =

    type T = EmailAddress of string with
        interface WrappedString.IWrappedString with
            member this.Value = let (EmailAddress s) = this in s

    let create =
        let canonicalize = WrappedString.singleLineTrimmed
        let isValid s =
            (WrappedString.lengthValidator 100 s) &&
            System.Text.RegularExpressions.Regex.IsMatch(s,@"^\S+@\S+\.\S+$")
        WrappedString.create canonicalize isValid EmailAddress
```

```
    /// Converts any wrapped string to an EmailAddress
    let convert s = WrappedString.apply create s

// ========================================
// ZipCode (not application specific)
// ========================================

module ZipCode =

    type T = ZipCode of string with
        interface WrappedString.IWrappedString with
            member this.Value = let (ZipCode s) = this in s

    let create =
        let canonicalize = WrappedString.singleLineTrimmed
        let isValid s =
            System.Text.RegularExpressions.Regex.IsMatch(s,@"^\d{5}$")
        WrappedString.create canonicalize isValid ZipCode

    /// Converts any wrapped string to a ZipCode
    let convert s = WrappedString.apply create s

// ========================================
// StateCode (not application specific)
// ========================================

module StateCode =

    type T = StateCode  of string with
        interface WrappedString.IWrappedString with
            member this.Value = let (StateCode  s) = this in s

    let create =
        let canonicalize = WrappedString.singleLineTrimmed
        let stateCodes = ["AZ";"CA";"NY"] //etc
        let isValid s =
            stateCodes |> List.exists ((=) s)

        WrappedString.create canonicalize isValid StateCode

    /// Converts any wrapped string to a StateCode
    let convert s = WrappedString.apply create s

// ========================================
// PostalAddress (not application specific)
// ========================================

module PostalAddress =

    type USPostalAddress =
        {
        Address1: WrappedString.String50;
        Address2: WrappedString.String50;
```

```
        City: WrappedString.String50;
        State: StateCode.T;
        Zip: ZipCode.T;
        }

    type UKPostalAddress =
        {
        Address1: WrappedString.String50;
        Address2: WrappedString.String50;
        Town: WrappedString.String50;
        PostCode: WrappedString.String50;   // todo
        }

    type GenericPostalAddress =
        {
        Address1: WrappedString.String50;
        Address2: WrappedString.String50;
        Address3: WrappedString.String50;
        Address4: WrappedString.String50;
        Address5: WrappedString.String50;
        }

    type T =
        | USPostalAddress of USPostalAddress
        | UKPostalAddress of UKPostalAddress
        | GenericPostalAddress of GenericPostalAddress

// ========================================
// PersonalName (not application specific)
// ========================================

module PersonalName =
    open WrappedString

    type T =
        {
        FirstName: String50;
        MiddleName: String50 option;
        LastName: String100;
        }

    /// create a new value
    let create first middle last =
        match (string50 first),(string100 last) with
        | Some f, Some l ->
            Some {
                FirstName = f;
                MiddleName = (string50 middle)
                LastName = l;
                }
        | _ ->
            None
```

```
    /// concat the names together
    /// and return a raw string
    let fullNameRaw personalName =
        let f = personalName.FirstName |> value
        let l = personalName.LastName |> value
        let names =
            match personalName.MiddleName with
            | None -> [| f; l |]
            | Some middle -> [| f; (value middle); l |]
        System.String.Join(" ", names)

    /// concat the names together
    /// and return None if too long
    let fullNameOption personalName =
        personalName |> fullNameRaw |> string100

    /// concat the names together
    /// and truncate if too long
    let fullNameTruncated personalName =
        // helper function
        let left n (s:string) =
            if (s.Length > n)
            then s.Substring(0,n)
            else s

        personalName
        |> fullNameRaw  // concat
        |> left 100     // truncate
        |> string100    // wrap
        |> Option.get   // this will always be ok
```

And now the application specific types.

```
// ========================================
// EmailContactInfo -- state machine
// ========================================

module EmailContactInfo =
    open System

    // UnverifiedData = just the EmailAddress
    type UnverifiedData = EmailAddress.T

    // VerifiedData = EmailAddress plus the time it was verified
    type VerifiedData = EmailAddress.T * DateTime

    // set of states
    type T =
        | UnverifiedState of UnverifiedData
        | VerifiedState of VerifiedData
```

```
    let create email =
        // unverified on creation
        UnverifiedState email

    // handle the "verified" event
    let verified emailContactInfo dateVerified =
        match emailContactInfo with
        | UnverifiedState email ->
            // construct a new info in the verified state
            VerifiedState (email, dateVerified)
        | VerifiedState _ ->
            // ignore
            emailContactInfo

    let sendVerificationEmail emailContactInfo =
        match emailContactInfo with
        | UnverifiedState email ->
            // send email
            printfn "sending email"
        | VerifiedState _ ->
            // do nothing
            ()

    let sendPasswordReset emailContactInfo =
        match emailContactInfo with
        | UnverifiedState email ->
            // ignore
            ()
        | VerifiedState _ ->
            // ignore
            printfn "sending password reset"

// ========================================
// PostalContactInfo -- state machine
// ========================================

module PostalContactInfo =
    open System

    // InvalidData = just the PostalAddress
    type InvalidData = PostalAddress.T

    // ValidData = PostalAddress plus the time it was verified
    type ValidData = PostalAddress.T * DateTime

    // set of states
    type T =
        | InvalidState of InvalidData
        | ValidState of ValidData

    let create address =
        // invalid on creation
```

```
        InvalidState address

    // handle the "validated" event
    let validated postalContactInfo dateValidated =
        match postalContactInfo with
        | InvalidState address ->
            // construct a new info in the valid state
            ValidState (address, dateValidated)
        | ValidState _ ->
            // ignore
            postalContactInfo

    let contactValidationService postalContactInfo =
        let dateIsTooLongAgo (d:DateTime) =
            d < DateTime.Today.AddYears(-1)

        match postalContactInfo with
        | InvalidState address ->
            printfn "contacting the address validation service"
        | ValidState (address,date) when date |> dateIsTooLongAgo  ->
            printfn "last checked a long time ago."
            printfn "contacting the address validation service again"
        | ValidState  _ ->
            printfn "recently checked. Doing nothing."

// ========================================
// ContactMethod and Contact
// ========================================

type ContactMethod =
    | Email of EmailContactInfo.T
    | PostalAddress of PostalContactInfo.T

type Contact =
    {
    Name: PersonalName.T;
    PrimaryContactMethod: ContactMethod;
    SecondaryContactMethods: ContactMethod list;
    }
```

Conclusion

Phew! The new code is much, much longer than the original code. Granted, it has a lot of supporting functions that were not needed in the original version, but even so it seems like a lot of extra work. So was it worth it?

I think the answer is yes. Here are some of the reasons why:

The new code is more explicit

If we look at the original example, there was no atomicity between fields, no validation rules, no length constraints, nothing to stop you updating flags in the wrong order, and so on.

The data structure was "dumb" and all the business rules were implicit in the application code. Chances are that the application would have lots of subtle bugs that might not even show up in unit tests. (*Are you sure the application reset the `IsEmailVerified` flag to false in every place the email address was updated?*)

On the other hand, the new code is extremely explicit about every little detail. If I stripped away everything but the types themselves, you would have a very good idea of what the business rules and domain constraints were.

The new code won't let you postpone error handling

Writing code that works with the new types means that you are forced to handle every possible thing that could go wrong, from dealing with a name that is too long, to failing to supply a contact method. And you have to do this up front at construction time. You can't postpone it till later.

Writing such error handling code can be annoying and tedious, but on the other hand, it pretty much writes itself. There is really only one way to write code that actually compiles with these types.

The new code is more likely to be correct

The *huge* benefit of the new code is that it is probably bug free. Without even writing any unit tests, I can be quite confident that a first name will never be truncated when written to a `varchar(50)` in a database, and that I can never accidentally send out a verification email twice.

And in terms of the code itself, many of the things that you as a developer have to remember to deal with (or forget to deal with) are completely absent. No null checks, no casting, no worrying about what the default should be in a `switch` statement. And if you like to use cyclomatic complexity as a code quality metric, you might note that there are only three `if` statements in the entire 350 odd lines.

A word of warning...

Finally, beware! Getting comfortable with this style of type-based design will have an insidious effect on you. You will start to develop paranoia whenever you see code that isn't typed strictly enough. (*How long should an email address be, exactly?*) and you will be unable to write the simplest python script without getting anxious. When this happens, you will have been fully inducted into the cult. Welcome!

If you liked this series, here is a slide deck that covers many of the same topics. There is a video as well (here)

Domain Driven Design with the F# type System -- F#unctional Londoners 2014 from **my slideshare page**

Algebraic type sizes and domain modelling

In this post, we'll look at how to calculate the "size", or cardinality, of an algebraic type, and see how this knowledge can help us with design decisions.

Getting started

I'm going to define the "size" of a type by thinking of it as a set, and counting the number of possible elements.

For example, there are two possible booleans, so the size of the `Boolean` type is two.

Is there a type with size one? Yes -- the `unit` type only has one value: `()`.

Is there a type with size zero? That is, is there a type that has no values at all? Not in F#, but in Haskell there is. It is called `Void`.

What about a type like this:

```
type ThreeState =
    | Checked
    | Unchecked
    | Unknown
```

What is its size? There are three possible values, so the size is three.

What about a type like this:

```
type Direction =
    | North
    | East
    | South
    | West
```

Obviously, four.

I think you get the idea!

Calculating the size of compound types

Let's look at calculating the sizes of compound types now. If you remember from the understanding F# types series, there are two kinds of algebraic types: "product" types such as tuples and records, and "sum" types, called discriminated unions in F#.

For example, let's say that we have a `Speed` as well as a `Direction`, and we combine them into a record type called `Velocity`:

```
type Speed =
    | Slow
    | Fast

type Velocity = {
    direction: Direction
    speed: Speed
    }
```

What is the size of `Velocity`?

Here's every possible value:

```
{direction=North; speed=Slow}; {direction=North; speed=Fast}
{direction=East;  speed=Slow}; {direction=East;  speed=Fast}
{direction=South; speed=Slow}; {direction=South; speed=Fast}
{direction=West;  speed=Slow}; {direction=West;  speed=Fast}
```

There are eight possible values, one for every possible combination of the two `Speed` values and the four `Direction` values.

We can generalize this into a rule:

- **RULE: The size of a product type is the *product* of the sizes of the component types.**

That is, given a record type like this:

```
type RecordType = {
    a : TypeA
    b : TypeB }
```

The size is calculated like this:

```
size(RecordType) = size(TypeA) * size(TypeB)
```

And similarly for a tuple:

```
type TupleType = TypeA * TypeB
```

The size is:

```
size(TupleType) = size(TypeA) * size(TypeB)
```

Sum types

Sum types can be analyzed the same way. Given a type `Movement` defined like this:

```
type Movement =
    | Moving of Direction
    | NotMoving
```

We can write out and count all the possibilities:

```
Moving North
Moving East
Moving South
Moving West
NotMoving
```

So, five in all. Which just happens to be `size(Direction) + 1`. Here's another fun one:

```
type ThingsYouCanSay =
    | Yes
    | Stop
    | Goodbye

type ThingsICanSay =
    | No
    | GoGoGo
    | Hello

type HelloGoodbye =
    | YouSay of ThingsYouCanSay
    | ISay of ThingsICanSay
```

Again, we can write out and count all the possibilities:

```
YouSay Yes
ISay No
YouSay Stop
ISay GoGoGo
YouSay Goodbye
ISay Hello
```

There are three possible values in the `YouSay` case, and three possible values in the `ISay` case, making six in all.

Again, we can make a general rule.

- **RULE: The size of a sum or union type is the *sum* of the sizes of the component types.**

That is, given a union type like this:

```
type SumType =
    | CaseA of TypeA
    | CaseB of TypeB
```

The size is calculated like this:

```
size(SumType) = size(TypeA) + size(TypeB)
```

Working with generic types

What happens if we throw generic types into the mix?

For example, what is the size of a type like this:

```
type Optional<'a> =
    | Something of 'a
    | Nothing
```

Well, the first thing to say is that `Optional<'a>` is not a *type* but a *type constructor*. `Optional<string>` is a type. `Optional<int>` is a type, but `Optional<'a>` isn't.

Nevertheless, we can still calculate its size by noting that `size(Optional<string>)` is just `size(string) + 1`, `size(Optional<int>)` is just `size(int) + 1`, and so on.

So we can say:

```
size(Optional<'a>) = size('a) + 1
```

Similarly, for a type with two generics like this:

```
type Either<'a,'b> =
    | Left of 'a
    | Right of 'b
```

we can say that its size can be calculated using the size of the generic components (using the "sum rule" above):

```
size(Either<'a,'b>) = size('a) + size('b)
```

Recursive types

What about a recursive type? Let's look at the simplest one, a linked list.

A linked list is either empty, or it has a cell with a tuple: a head and a tail. The head is an `'a` and the tail is another list. Here's the definition:

```
type LinkedList<'a> =
    | Empty
    | Node of head:'a * tail:LinkedList<'a>
```

To calculate the size, let's assign some names to the various components:

```
let S = size(LinkedList<'a>)
let N = size('a)
```

Now we can write:

```
S =
    1         // Size of "Empty" case
    +         // Union type
    N * S     // Size of "Cell" case using tuple size calculation
```

Let's play with this formula a bit. We start with:

```
S = 1 + (N * S)
```

and let's substitute the last S with the formula to get:

```
S = 1 + (N * (1 + (N * S)))
```

If we clean this up, we get:

```
S = 1 + N + (N^2 * S)
```

(where `N^2` means "N squared")

Let's substitute the last S with the formula again:

```
S = 1 + N + (N^2 * (1 + (N * S)))
```

and clean up again:

```
S = 1 + N + N^2 + (N^3 * S)
```

You can see where this is going! The formula for `S` can be expanded out indefinitely to be:

```
S = 1 + N + N^2 + N^3 + N^4 + N^5 + ...
```

How can we interpret this? Well, we can say that a list is a union of the following cases:

- an empty list(size = 1)
- a one element list (size = N)
- a two element list (size = N x N)
- a three element list (size = N x N x N)
- and so on.

And this formula has captured that.

As an aside, you can calculate `S` directly using the formula `S = 1/(1-N)`, which means that a list of `Direction` (size=4) has size "-1/3". Hmmm, that's strange! It reminds me of this "-1/12" video.

Calculating the size of functions

What about functions? Can they be sized?

Yes, all we need to do is write down every possible implementation and count them. Easy!

For example, say that we have a function `SuitColor` that maps a card `Suit` to a `Color`, red or black.

```
type Suit = Heart | Spade | Diamond | Club
type Color = Red | Black

type SuitColor = Suit -> Color
```

One implementation would be to return red, no matter what suit was provided:

```
(Heart -> Red); (Spade -> Red); (Diamond -> Red); (Club -> Red)
```

Another implementation would be to return red for all suits except `Club`:

```
(Heart -> Red); (Spade -> Red); (Diamond -> Red); (Club -> Black)
```

In fact we can write down all 16 possible implementations of this function:

```
(Heart -> Red); (Spade -> Red); (Diamond -> Red); (Club -> Red)
(Heart -> Red); (Spade -> Red); (Diamond -> Red); (Club -> Black)
(Heart -> Red); (Spade -> Red); (Diamond -> Black); (Club -> Red)
(Heart -> Red); (Spade -> Red); (Diamond -> Black); (Club -> Black)

(Heart -> Red); (Spade -> Black); (Diamond -> Red); (Club -> Red)
(Heart -> Red); (Spade -> Black); (Diamond -> Red); (Club -> Black)  // the right one!
(Heart -> Red); (Spade -> Black); (Diamond -> Black); (Club -> Red)
(Heart -> Red); (Spade -> Black); (Diamond -> Black); (Club -> Black)

(Heart -> Black); (Spade -> Red); (Diamond -> Red); (Club -> Red)
(Heart -> Black); (Spade -> Red); (Diamond -> Red); (Club -> Black)
(Heart -> Black); (Spade -> Red); (Diamond -> Black); (Club -> Red)
(Heart -> Black); (Spade -> Red); (Diamond -> Black); (Club -> Black)

(Heart -> Black); (Spade -> Black); (Diamond -> Red); (Club -> Red)
(Heart -> Black); (Spade -> Black); (Diamond -> Red); (Club -> Black)
(Heart -> Black); (Spade -> Black); (Diamond -> Black); (Club -> Red)
(Heart -> Black); (Spade -> Black); (Diamond -> Black); (Club -> Black)
```

Another way to think of it is that we can define a record type where each value represents a particular implementation: which color do we return for a `Heart` input, which color do we return for a `Spade` input, and so on.

The type definition for the implementations of `SuitColor` would therefore look like this:

```
type SuitColorImplementation = {
    Heart : Color
    Spade : Color
    Diamond : Color
    Club : Color }
```

What is the size of this record type?

```
size(SuitColorImplementation) = size(Color) * size(Color) * size(Color) * size(Color)
```

There are four `size(Color)` here. In other words, there is one `size(Color)` for every input, so we could write this as:

```
size(SuitColorImplementation) = size(Color) to the power of size(Suit)
```

In general, then, given a function type:

```
type Function<'input,'output> = 'input -> 'output
```

The size of the function is `size(output type)` to the power of `size(input type)` :

```
size(Function) =  size(output) ^ size(input)
```

Lets codify that into a rule too:

- **RULE: The size of a function type is `size(output type)` to the power of `size(input type)` .**

Converting between types

All right, that is all very interesting, but is it *useful*?

Yes, I think it is. I think that understanding sizes of types like this helps us design conversions from one type to another, which is something we do a lot of!

Let's say that we have a union type and a record type, both representing a yes/no answer:

```
type YesNoUnion =
    | Yes
    | No

type YesNoRecord = {
    isYes: bool }
```

How can we map between them?

They both have size=2, so we should be able to map each value in one type to the other, and vice versa:

```
let toUnion yesNoRecord =
    if yesNoRecord.isYes then
        Yes
    else
        No

let toRecord yesNoUnion =
    match yesNoUnion with
    | Yes -> {isYes = true}
    | No ->  {isYes = false}
```

This is what you might call a "lossless" conversion. If you round-trip the conversion, you can recover the original value. Mathematicians would call this an *isomorphism* (from the Greek "equal shape").

What about another example? Here's a type with three cases, yes, no, and maybe.

```
type YesNoMaybe =
    | Yes
    | No
    | Maybe
```

Can we losslessly convert this to this type?

```
type YesNoOption = { maybeIsYes: bool option }
```

Well, what is the size of an `option` ? One plus the size of the inner type, which in this case is a `bool` . So `size(YesNoOption)` is also three.

Here are the conversion functions:

```
let toYesNoMaybe yesNoOption =
    match yesNoOption.maybeIsYes with
    | None -> Maybe
    | Some b -> if b then Yes else No

let toYesNoOption yesNoMaybe =
    match yesNoMaybe with
    | Yes ->   {maybeIsYes = Some true}
    | No ->    {maybeIsYes = Some false}
    | Maybe -> {maybeIsYes = None}
```

So we can make a rule:

- **RULE: If two types have the same size, you can create a pair of lossless conversion functions**

Let's try it out. Here's a `Nibble` type and a `TwoNibbles` type:

```
type Nibble = {
    bit1: bool
    bit2: bool
    bit3: bool
    bit4: bool }

type TwoNibbles = {
    high: Nibble
    low: Nibble }
```

Can we convert `TwoNibbles` to a `byte` and back?

The size of `Nibble` is `2 x 2 x 2 x 2` = 16 (using the product size rule), and the size of `TwoNibbles` is size(Nibble) x size(Nibble), or `16 x 16`, which is 256.

So yes, we can convert from `TwoNibbles` to a `byte` and back.

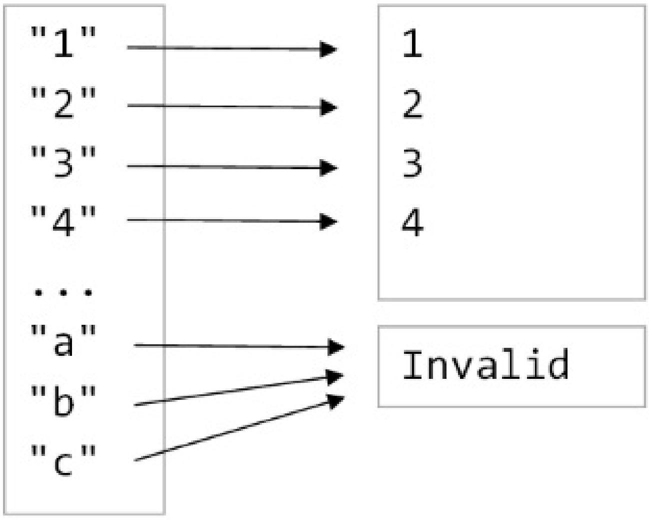
Lossy conversions

What happens if the types are different sizes?

If the target type is "larger" than the source type, then you can always map without loss, but if the target type is "smaller" than the source type, you have a problem.

For example, the `int` type is smaller than the `string` type. You can convert an `int` to a `string` accurately, but you can't convert a `string` to an `int` easily.

If you *do* want to map a string to an int, then some of the non-integer strings will have to be mapped to a special, non-integer value in the target type:

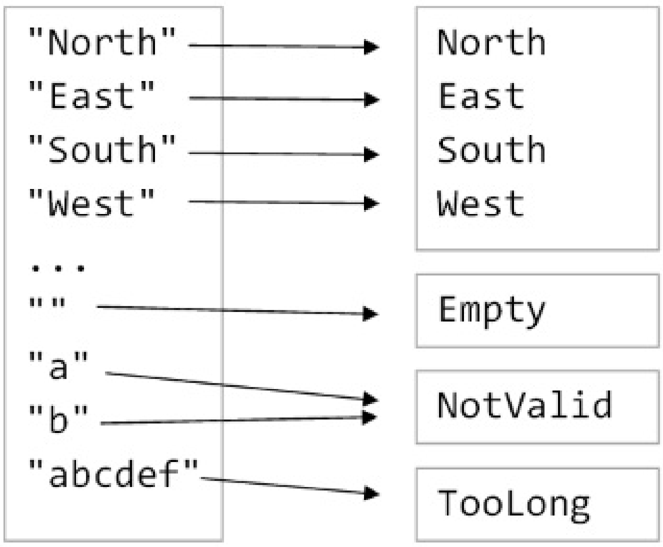


In other words we know from the sizes that the target type can't just be an `int` type, it must be an `int + 1` type. In other words, an Option type!

Interestingly, the `Int32.TryParse` function in the BCL returns two values, a success/failure `bool` and the parsed result as an `int`. In other words, a tuple `bool * int`.

The size of that tuple is `2 x int`, many more values that are really needed. Option types ftw!

Now let's say we are converting from a `string` to a `Direction`. Some strings are valid, but most of them are not. But this time, instead of having one invalid case, let's also say that we want to distinguish between empty inputs, inputs that are too long, and other invalid inputs.



We can't model the target with an Option any more, so let's design a custom type that contains all seven cases:

```
type StringToDirection_V1 =
    | North
    | East
    | South
    | West
    | Empty
    | NotValid
    | TooLong
```

But this design mixes up successful conversions and failed conversions. Why not separate them?

```
type Direction =
    | North
    | East
    | South
    | West

type ConversionFailure =
    | Empty
    | NotValid
    | TooLong

type StringToDirection_V2 =
    | Success of Direction
    | Failure of ConversionFailure
```

What is the size of `StringToDirection_V2`?

There are 4 choices of `Direction` in the `Success` case, and three choices of `ConversionFailure` in the `Failure` case, so the total size is seven, just as in the first version.

In other words, both of these designs are *equivalent* and we can use either one.

Personally, I prefer version 2, but if we had version 1 in our legacy code, the good news is that we can losslessly convert from version 1 to version 2 and back again. Which in turn means that we can safely refactor to version 2 if we need to.

Designing the core domain

Knowing that different types can be losslessly converted allows you to tweak your domain designs as needed.

For example, this type:

```
type Something_V1 =
    | CaseA1 of TypeX * TypeY
    | CaseA2 of TypeX * TypeZ
```

can be losslessly converted to this one:

```
type Inner =
    | CaseA1 of TypeY
    | CaseA2 of TypeZ

type Something_V2 =
    TypeX * Inner
```

or this one:

```
type Something_V3 = {
    x: TypeX
    inner: Inner }
```

Here's a real example:

- You have a website where some users are registered and some are not.
- For all users, you have a session id
- For registered users only, you have extra information

We could model that requirement like this:

```
module Customer_V1 =

    type UserInfo = {name:string} //etc
    type SessionId = SessionId of int

    type WebsiteUser =
        | RegisteredUser of SessionId * UserInfo
        | GuestUser of SessionId
```

or alternatively, we can pull the common `SessionId` up to a higher level like this:

```
module Customer_V2 =

    type UserInfo = {name:string} //etc
    type SessionId = SessionId of int

    type WebsiteUserInfo =
        | RegisteredUser of UserInfo
        | GuestUser

    type WebsiteUser = {
        sessionId : SessionId
        info: WebsiteUserInfo }
```

Which is better? In one sense, they are both the "same", but obviously the best design depends on the usage pattern.

- If you care more about the type of user than the session id, then version 1 is better.
- If you are constantly looking at the session id without caring about the type of user, then version 2 is better.

The nice thing about knowing that they are isomorphic is that you can define *both* types if you like, use them in different contexts, and losslessly map between them as needed.

Interfacing with the outside world

We have all these nice domain types like `Direction` or `WebsiteUser` but at some point we need to interface with the outside world -- store them in a database, receive them as JSON, etc.

The problem is that the outside world does not have a nice type system! Everything tends to be primitives: strings, ints and bools.

Going from our domain to the outside world means going from types with a "small" set of values to types with a "large" set of values, which we can do straightforwardly. But coming in from the outside world into our domain means going from a "large" set of values to a "small" set of values, which requires validation and error cases.

For example, a domain type might look like this:

```
type DomainCustomer = {
    Name: String50
    Email: EmailAddress
    Age: PositiveIntegerLessThan130 }
```

The values are constrained: max 50 chars for the name, a validated email, an age which is between 1 and 129.

On the other hand, the DTO type might look like this:

```
type CustomerDTO = {
    Name: string
    Email: string
    Age: int }
```

The values are unconstrained: any string for the name, a unvalidated email, an age that can be any of 2^32 different values, including negative ones.

This means that we *cannot* create a `CustomerDTO` to `DomainCustomer` mapping. We *have* to have at least one other value (`DomainCustomer + 1`) to map the invalid inputs onto, and preferably more to document the various errors.

This leads naturally to the `Success/Failure` model as described in my functional error handling talk,

The final version of the mapping would then be from a `CustomerDTO` to a `SuccessFailure<DomainCustomer>` or similar.

So that leads to the final rule:

- **RULE: Trust no one. If you import data from an external source, be sure to handle invalid input.**

If we take this rule seriously, it has some knock on effects, such as:

- Never try to deserialize directly to a domain type (e.g. no ORMs), only to DTO types.
- Always validate every record you read from a database or other "trusted" source.

You might think that having everything wrapped in a `Success/Failure` type can get annoying, and this is true (!), but there are ways to make this easier. See this post for example.

Further reading

The "algebra" of algebraic data types is well known. There is a good recent summary in "The algebra (and calculus!) of algebraic data types" and a series by Chris Taylor.

And after I wrote this, I was pointed to two similar posts:

- One by Tomas Petricek with almost the same content!

- One by Bartosz Milewski in his series on category theory.

As some of those posts mention, you can do strange things with these type formulas, such as differentiate them!

If you like academic papers, you can read the original discussion of derivatives in "The Derivative of a Regular Type is its Type of One-Hole Contexts"(PDF) by Conor McBride from 2001, and a follow up in "Differentiating Data Structures"(PDF) [Abbott, Altenkirch, Ghani, and McBride, 2005].

Summary

This might not be the most exciting topic in the world, but I've found this approach both interesting and useful, and I wanted to share it with you.

Let me know what you think. Thanks for reading!

Thirteen ways of looking at a turtle

> This post is part of the F# Advent Calendar in English 2015 project. Check out all the other great posts there! And special thanks to Sergey Tihon for organizing this.

I was discussing how to implement a simple turtle graphics system some time ago, and it struck me that, because the turtle requirements are so simple and so well known, it would make a great basis for demonstrating a range of different techniques.

So, in this two part mega-post, I'll stretch the turtle model to the limit while demonstrating things like: partial application, validation with Success/Failure results, the concept of "lifting", agents with message queues, dependency injection, the State monad, event sourcing, stream processing, and finally a custom interpreter!

Without further ado then, I hereby present thirteen different ways of implementing a turtle:

- Way 1. A basic object-oriented approach, in which we create a class with mutable state.
- Way 2. A basic functional approach, in which we create a module of functions with immutable state.
- Way 3. An API with a object-oriented core, in which we create an object-oriented API that calls a stateful core class.
- Way 4. An API with a functional core, in which we create an stateful API that uses stateless core functions.
- Way 5. An API in front of an agent, in which we create an API that uses a message queue to communicate with an agent.
- Way 6. Dependency injection using interfaces, in which we decouple the implementation from the API using an interface or record of functions.
- Way 7. Dependency injection using functions, in which we decouple the implementation from the API by passing a function parameter.
- Way 8. Batch processing using a state monad, in which we create a special "turtle workflow" computation expression to track state for us.
- Way 9. Batch processing using command objects, in which we create a type to represent a turtle command, and then process a list of commands all at once.
- Interlude: Conscious decoupling with data types. A few notes on using data vs. interfaces for decoupling.
- Way 10. Event sourcing, in which state is built from a list of past events.
- Way 11. Functional Retroactive Programming (stream processing), in which business logic is based on reacting to earlier events.
- Episode V: The Turtle Strikes Back, in which the turtle API changes so that some commands may fail.

- Way 12. Monadic control flow, in which we make decisions in the turtle workflow based on results from earlier commands.
- Way 13. A turtle interpreter, in which we completely decouple turtle programming from turtle implementation, and nearly encounter the free monad.
- Review of all the techniques used.

and 2 bonus ways for the extended edition:

- Way 14. Abstract Data Turtle, in which we encapsulate the details of a turtle implementation by using an Abstract Data Type.
- Way 15. Capability-based Turtle, in which we control what turtle functions are available to a client, based on the current state of the turtle.

All source code for this post is available on github.

The requirements for a Turtle

A turtle supports four instructions:

- Move some distance in the current direction.
- Turn a certain number of degrees clockwise or anticlockwise.
- Put the pen down or up. When the pen is down, moving the turtle draws a line.
- Set the pen color (one of black, blue or red).

These requirements lead naturally to some kind of "turtle interface" like this:

- `Move aDistance`
- `Turn anAngle`
- `PenUp`
- `PenDown`
- `SetColor aColor`

All of the following implementations will be based on this interface or some variant of it.

Note that the turtle must convert these instructions to drawing lines on a canvas or other graphics context. So the implementation will probably need to keep track of the turtle position and current state somehow.

Common code

Before we start implementing, let's get some common code out of the way.

First, we'll need some types to represent distances, angles, the pen state, and the pen colors.

```
/// An alias for a float
type Distance = float

/// Use a unit of measure to make it clear that the angle is in degrees, not radians
type [<Measure>] Degrees

/// An alias for a float of Degrees
type Angle  = float<Degrees>

/// Enumeration of available pen states
type PenState = Up | Down

/// Enumeration of available pen colors
type PenColor = Black | Red | Blue
```

and we'll also need a type to represent the position of the turtle:

```
/// A structure to store the (x,y) coordinates
type Position = {x:float; y:float}
```

We'll also need a helper function to calculate a new position based on moving a certain distance at a certain angle:

```
// round a float to two places to make it easier to read
let round2 (flt:float) = Math.Round(flt,2)

/// calculate a new position from the current position given an angle and a distance
let calcNewPosition (distance:Distance) (angle:Angle) currentPos =
    // Convert degrees to radians with 180.0 degrees = 1 pi radian
    let angleInRads = angle * (Math.PI/180.0) * 1.0<1/Degrees>
    // current pos
    let x0 = currentPos.x
    let y0 = currentPos.y
    // new pos
    let x1 = x0 + (distance * cos angleInRads)
    let y1 = y0 + (distance * sin angleInRads)
    // return a new Position
    {x=round2 x1; y=round2 y1}
```

Let's also define the initial state of a turtle:

```
/// Default initial state
let initialPosition,initialColor,initialPenState =
    {x=0.0; y=0.0}, Black, Down
```

And a helper that pretends to draw a line on a canvas:

```
let dummyDrawLine log oldPos newPos color =
    // for now just log it
    log (sprintf "...Draw line from (%0.1f,%0.1f) to (%0.1f,%0.1f) using %A" oldPos.x
oldPos.y newPos.x newPos.y color)
```

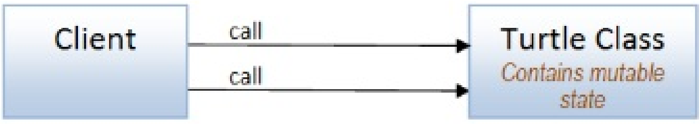
Now we're ready for the first implementation!

---

1. Basic OO -- A class with mutable state

In this first design, we will use an object-oriented approach and represent the turtle with a simple class.

- The state will be stored in local fields (`currentPosition` , `currentAngle` , etc) that are mutable.
- We will inject a logging function `log` so that we can monitor what happens.



And here's the complete code, which should be self-explanatory:

```
type Turtle(log) =

    let mutable currentPosition = initialPosition
    let mutable currentAngle = 0.0<Degrees>
    let mutable currentColor = initialColor
    let mutable currentPenState = initialPenState

    member this.Move(distance) =
        log (sprintf "Move %0.1f" distance)
        // calculate new position
        let newPosition = calcNewPosition distance currentAngle currentPosition
        // draw line if needed
        if currentPenState = Down then
            dummyDrawLine log currentPosition newPosition currentColor
        // update the state
        currentPosition <- newPosition

    member this.Turn(angle) =
        log (sprintf "Turn %0.1f" angle)
        // calculate new angle
        let newAngle = (currentAngle + angle) % 360.0<Degrees>
        // update the state
        currentAngle <- newAngle

    member this.PenUp() =
        log "Pen up"
        currentPenState <- Up

    member this.PenDown() =
        log "Pen down"
        currentPenState <- Down

    member this.SetColor(color) =
        log (sprintf "SetColor %A" color)
        currentColor <- color
```

Calling the turtle object

The client code instantiates the turtle and talks to it directly:

```
/// Function to log a message
let log message =
    printfn "%s" message

let drawTriangle() =
    let turtle = Turtle(log)
    turtle.Move 100.0
    turtle.Turn 120.0<Degrees>
    turtle.Move 100.0
    turtle.Turn 120.0<Degrees>
    turtle.Move 100.0
    turtle.Turn 120.0<Degrees>
    // back home at (0,0) with angle 0
```

The logged output of `drawTriangle()` is:

```
Move 100.0
...Draw line from (0.0,0.0) to (100.0,0.0) using Black
Turn 120.0
Move 100.0
...Draw line from (100.0,0.0) to (50.0,86.6) using Black
Turn 120.0
Move 100.0
...Draw line from (50.0,86.6) to (0.0,0.0) using Black
Turn 120.0
```

Similarly, here's the code to draw a polygon:

```
let drawPolygon n =
    let angle = 180.0 - (360.0/float n)
    let angleDegrees = angle * 1.0<Degrees>
    let turtle = Turtle(log)

    // define a function that draws one side
    let drawOneSide() =
        turtle.Move 100.0
        turtle.Turn angleDegrees

    // repeat for all sides
    for i in [1..n] do
        drawOneSide()
```

Note that `drawOneSide()` does not return anything -- all the code is imperative and stateful. Compare this to the code in the next example, which takes a pure functional approach.

Advantages and disadvantages

So what are the advantages and disadvantages of this simple approach?

Advantages

- It's very easy to implement and understand.

Disadvantages

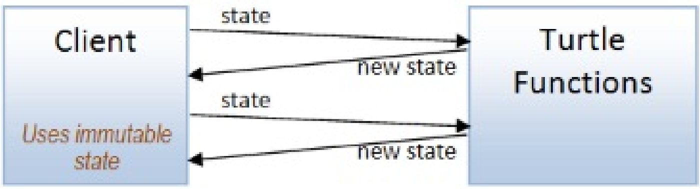
- The stateful code is harder to test. We have to put an object into a known state state before testing, which is simple in this case, but can be long-winded and error-prone for more complex objects.
- The client is coupled to a particular implementation. No interfaces here! We'll look at using interfaces shortly.

The source code for this version is available here (turtle class) and here (client).

2: Basic FP - A module of functions with immutable state

The next design will use a pure, functional approach. An immutable `TurtleState` is defined, and then the various turtle functions accept a state as input and return a new state as output.

In this approach then, the client is responsible for keeping track of the current state and passing it into the next function call.



Here's the definition of `TurtleState` and the values for the initial state:

```
module Turtle =

    type TurtleState = {
        position : Position
        angle : float<Degrees>
        color : PenColor
        penState : PenState
    }

    let initialTurtleState = {
        position = initialPosition
        angle = 0.0<Degrees>
        color = initialColor
        penState = initialPenState
    }
```

And here are the "api" functions, all of which take a state parameter and return a new state:

```
module Turtle =

    // [state type snipped]

    let move log distance state =
        log (sprintf "Move %0.1f" distance)
        // calculate new position
        let newPosition = calcNewPosition distance state.angle state.position
        // draw line if needed
        if state.penState = Down then
            dummyDrawLine log state.position newPosition state.color
        // update the state
        {state with position = newPosition}

    let turn log angle state =
        log (sprintf "Turn %0.1f" angle)
        // calculate new angle
        let newAngle = (state.angle + angle) % 360.0<Degrees>
        // update the state
        {state with angle = newAngle}

    let penUp log state =
        log "Pen up"
        {state with penState = Up}

    let penDown log state =
        log "Pen down"
        {state with penState = Down}

    let setColor log color state =
        log (sprintf "SetColor %A" color)
        {state with color = color}
```

Note that the `state` is always the last parameter -- this makes it easier to use the "piping" idiom.

Using the turtle functions

The client now has to pass in both the `log` function and the `state` to every function, every time!

We can eliminate the need to pass in the log function by using partial application to create new versions of the functions with the logger baked in:

```
/// Function to log a message
let log message =
    printfn "%s" message

// versions with log baked in (via partial application)
let move = Turtle.move log
let turn = Turtle.turn log
let penDown = Turtle.penDown log
let penUp = Turtle.penUp log
let setColor = Turtle.setColor log
```

With these simpler versions, the client can just pipe the state through in a natural way:

```
let drawTriangle() =
    Turtle.initialTurtleState
    |> move 100.0
    |> turn 120.0<Degrees>
    |> move 100.0
    |> turn 120.0<Degrees>
    |> move 100.0
    |> turn 120.0<Degrees>
    // back home at (0,0) with angle 0
```

When it comes to drawing a polygon, it's a little more complicated, as we have to "fold" the state through the repetitions for each side:

```
let drawPolygon n =
    let angle = 180.0 - (360.0/float n)
    let angleDegrees = angle * 1.0<Degrees>

    // define a function that draws one side
    let oneSide state sideNumber =
        state
        |> move 100.0
        |> turn angleDegrees

    // repeat for all sides
    [1..n]
    |> List.fold oneSide Turtle.initialTurtleState
```

Advantages and disadvantages

What are the advantages and disadvantages of this purely functional approach?

Advantages

- Again, it's very easy to implement and understand.
- The stateless functions are easier to test. We always provide the current state as input, so there is no setup needed to get an object into a known state.
- Because there is no global state, the functions are modular and can be reused in other contexts (as we'll see later in this post).

Disadvantages

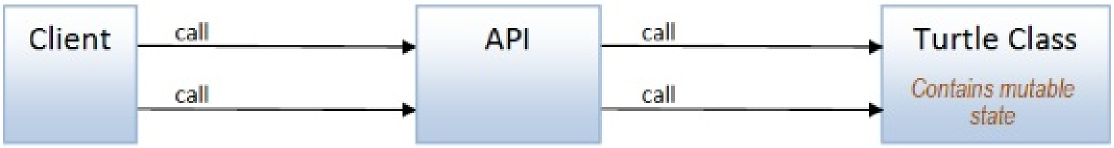
- As before, the client is coupled to a particular implementation.
- The client has to keep track of the state (but some solutions to make this easier are shown later in this post).

The source code for this version is available here (turtle functions) and here (client).

3: An API with a object-oriented core

Let's hide the client from the implementation using an API!

In this case, the API will be string based, with text commands such as `"move 100"` or `"turn 90"`. The API must validate these commands and turn them into method calls on the turtle (we'll use the OO approach of a stateful `Turtle` class again).



If the command is *not* valid, the API must indicate that to the client. Since we are using an OO approach, we'll do this by throwing a `TurtleApiException` containing a string, like this.

```
exception TurtleApiException of string
```

Next we need some functions that validate the command text:

```
// convert the distance parameter to a float, or throw an exception
let validateDistance distanceStr =
    try
        float distanceStr
    with
    | ex ->
        let msg = sprintf "Invalid distance '%s' [%s]" distanceStr  ex.Message
        raise (TurtleApiException msg)

// convert the angle parameter to a float<Degrees>, or throw an exception
let validateAngle angleStr =
    try
        (float angleStr) * 1.0<Degrees>
    with
    | ex ->
        let msg = sprintf "Invalid angle '%s' [%s]" angleStr ex.Message
        raise (TurtleApiException msg)

// convert the color parameter to a PenColor, or throw an exception
let validateColor colorStr =
    match colorStr with
    | "Black" -> Black
    | "Blue" -> Blue
    | "Red" -> Red
    | _ ->
        let msg = sprintf "Color '%s' is not recognized" colorStr
        raise (TurtleApiException msg)
```

With these in place, we can create the API.

The logic for parsing the command text is to split the command text into tokens and then match the first token to `"move"` , `"turn"` , etc.

Here's the code:

```
type TurtleApi() =

    let turtle = Turtle(log)

    member this.Exec (commandStr:string) =
        let tokens = commandStr.Split(' ') |> List.ofArray |> List.map trimString
        match tokens with
        | [ "Move"; distanceStr ] ->
            let distance = validateDistance distanceStr
            turtle.Move distance
        | [ "Turn"; angleStr ] ->
            let angle = validateAngle angleStr
            turtle.Turn angle
        | [ "Pen"; "Up" ] ->
            turtle.PenUp()
        | [ "Pen"; "Down" ] ->
            turtle.PenDown()
        | [ "SetColor"; colorStr ] ->
            let color = validateColor colorStr
            turtle.SetColor color
        | _ ->
            let msg = sprintf "Instruction '%s' is not recognized" commandStr
            raise (TurtleApiException msg)
```

Using the API

Here's how `drawPolygon` is implemented using the `TurtleApi` class:

```
let drawPolygon n =
    let angle = 180.0 - (360.0/float n)
    let api = TurtleApi()

    // define a function that draws one side
    let drawOneSide() =
        api.Exec "Move 100.0"
        api.Exec (sprintf "Turn %f" angle)

    // repeat for all sides
    for i in [1..n] do
        drawOneSide()
```

You can see that the code is quite similar to the earlier OO version, with the direct call `turtle.Move 100.0` being replaced with the indirect API call `api.Exec "Move 100.0"`.

Now if we trigger an error with a bad command such as `api.Exec "Move bad"`, like this:

```
let triggerError() =
    let api = TurtleApi()
    api.Exec "Move bad"
```

then the expected exception is thrown:

```
Exception of type 'TurtleApiException' was thrown.
```

Advantages and disadvantages

What are the advantages and disadvantages of an API layer like this?

- The turtle implementation is now hidden from the client.
- An API at a service boundary supports validation and can be extended to support monitoring, internal routing, load balancing, etc.

Disadvantages

- The API is coupled to a particular implementation, even though the client isn't.
- The system is very stateful. Even though the client does not know about the implementation behind the API, the client is still indirectly coupled to the inner core via shared state which in turn can make testing harder.

The source code for this version is available here.

4: An API with a functional core

An alternative approach for this scenario is to use a hybrid design, where the core of the application consists of pure functions, while the boundaries are imperative and stateful.

This approach has been named "Functional Core/Imperative Shell" by Gary Bernhardt.

Applied to our API example, the API layer uses only pure turtle functions, but the API layer manages the state (rather than the client) by storing a mutable turtle state.

Also, to be more functional, the API will not throw exceptions if the command text is not valid, but instead will return a `Result` value with `Success` and `Failure` cases, where the `Failure` case is used for any errors. (See my talk on the functional approach to error handling for a more in depth discussion of this technique).



Let's start by implementing the API class. This time it contains a `mutable` turtle state:

```
type TurtleApi() =

    let mutable state = initialTurtleState

    /// Update the mutable state value
    let updateState newState =
        state <- newState
```

The validation functions no longer throw an exception, but return `Success` or `Failure` :

```
let validateDistance distanceStr =
    try
        Success (float distanceStr)
    with
    | ex ->
        Failure (InvalidDistance distanceStr)
```

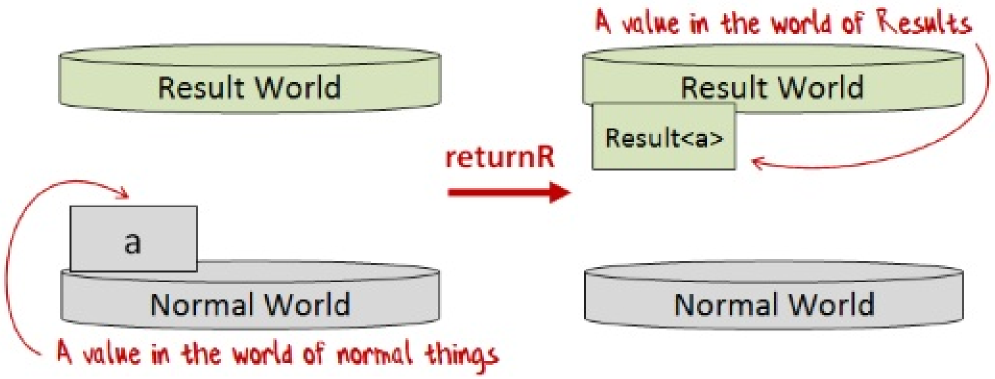
The error cases are documented in their own type:

```
type ErrorMessage =
    | InvalidDistance of string
    | InvalidAngle of string
    | InvalidColor of string
    | InvalidCommand of string
```

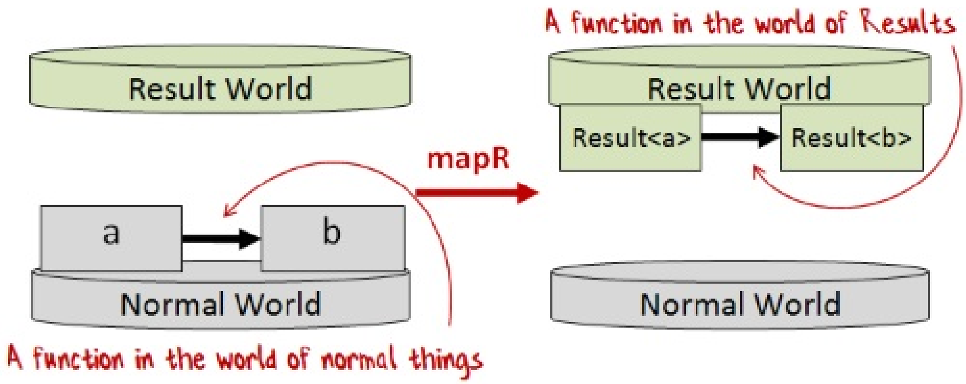
Now because the validation functions now return a `Result<Distance>` rather than a "raw" distance, the `move` function needs to be lifted to the world of `Results` , as does the current state.

There are three functions that we will use when working with `Result` s: `returnR` , `mapR` and `lift2R` .

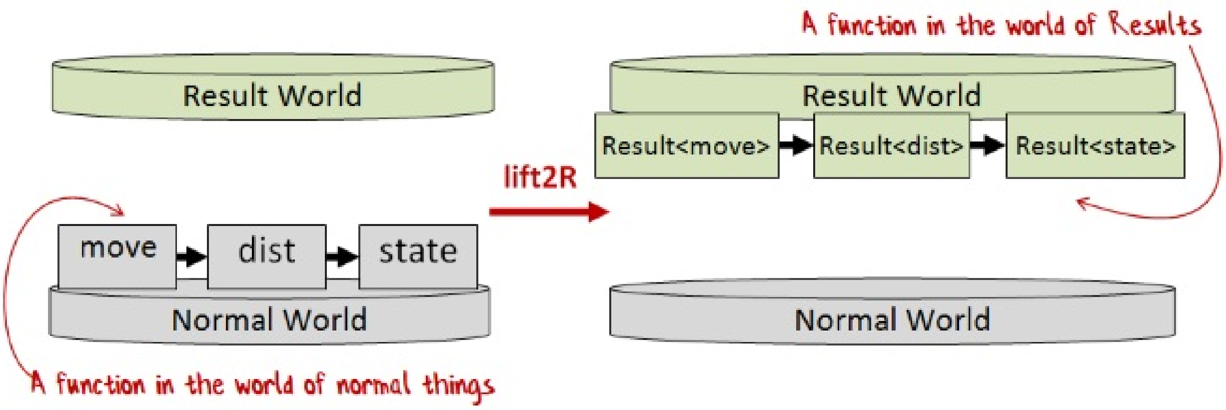
- `returnR` transforms a "normal" value into a value in the world of Results:



- `mapR` transforms a "normal" one-parameter function into a one-parameter function in the world of Results:



- `lift2R` transforms a "normal" two-parameter function into a two-parameter function in the world of Results:



As an example, with these helper functions, we can turn the normal `move` function into a function in the world of Results:

- The distance parameter is already in `Result` world
- The state parameter is lifted into `Result` world using `returnR`
- The `move` function is lifted into `Result` world using `lift2R`

```
// lift current state to Result
let stateR = returnR state

// get the distance as a Result
let distanceR = validateDistance distanceStr

// call "move" lifted to the world of Results
lift2R move distanceR stateR
```

(For more details on lifting functions to `Result` world, see the post on "lifting" in general)

Here's the complete code for `Exec` :

```
/// Execute the command string, and return a Result
/// Exec : commandStr:string -> Result<unit,ErrorMessage>
member this.Exec (commandStr:string) =
    let tokens = commandStr.Split(' ') |> List.ofArray |> List.map trimString

    // lift current state to Result
    let stateR = returnR state

    // calculate the new state
    let newStateR =
        match tokens with
        | [ "Move"; distanceStr ] ->
            // get the distance as a Result
            let distanceR = validateDistance distanceStr

            // call "move" lifted to the world of Results
            lift2R move distanceR stateR

        | [ "Turn"; angleStr ] ->
            let angleR = validateAngle angleStr
            lift2R turn angleR stateR

        | [ "Pen"; "Up" ] ->
            returnR (penUp state)

        | [ "Pen"; "Down" ] ->
            returnR (penDown state)

        | [ "SetColor"; colorStr ] ->
            let colorR = validateColor colorStr
            lift2R setColor colorR stateR

        | _ ->
            Failure (InvalidCommand commandStr)

    // Lift `updateState` into the world of Results and
    // call it with the new state.
    mapR updateState newStateR

    // Return the final result (output of updateState)
```

Using the API

The API returns a `Result` , so the client can no longer call each function in sequence, as we need to handle any errors coming from a call and abandon the rest of the steps.

To make our lives easier, we'll use a `result` computation expression (or workflow) to chain the calls and preserve the imperative "feel" of the OO version.

```
let drawTriangle() =
    let api = TurtleApi()
    result {
        do! api.Exec "Move 100"
        do! api.Exec "Turn 120"
        do! api.Exec "Move 100"
        do! api.Exec "Turn 120"
        do! api.Exec "Move 100"
        do! api.Exec "Turn 120"
        }
```

The source code for the `result` computation expression is available here.

Similarly, for the `drawPolygon` code, we can create a helper to draw one side and then call it `n` times inside a `result` expression.

```
let drawPolygon n =
    let angle = 180.0 - (360.0/float n)
    let api = TurtleApi()

    // define a function that draws one side
    let drawOneSide() = result {
        do! api.Exec "Move 100.0"
        do! api.Exec (sprintf "Turn %f" angle)
        }

    // repeat for all sides
    result {
        for i in [1..n] do
            do! drawOneSide()
    }
```

The code looks imperative, but is actually purely functional, as the returned `Result` values are being handled transparently by the `result` workflow.

Advantages and disadvantages

Advantages

- The same as for the OO version of an API -- the turtle implementation is hidden from the client, validation can be done, etc.
- The only stateful part of the system is at the boundary. The core is stateless which makes testing easier.

Disadvantages

- The API is still coupled to a particular implementation.

The source code for this version is available here (api helper functions) and here (API and client).

5: An API in front of an agent

In this design, an API layer communicates with a `TurtleAgent` via a message queue and the client talks to the API layer as before.



There are no mutables in the API (or anywhere). The `TurtleAgent` manages state by storing the current state as a parameter in the recursive message processing loop.

Now because the `TurtleAgent` has a typed message queue, where all messages are the same type, we must combine all possible commands into a single discriminated union type (`TurtleCommand`).

```
type TurtleCommand =
    | Move of Distance
    | Turn of Angle
    | PenUp
    | PenDown
    | SetColor of PenColor
```

The agent implementation is similar to the previous ones, but rather than exposing the turtle functions directly, we now do pattern matching on the incoming command to decide which function to call:

```
type TurtleAgent() =

    /// Function to log a message
    let log message =
        printfn "%s" message

    // logged versions
    let move = Turtle.move log
    let turn = Turtle.turn log
    let penDown = Turtle.penDown log
    let penUp = Turtle.penUp log
    let setColor = Turtle.setColor log

    let mailboxProc = MailboxProcessor.Start(fun inbox ->
        let rec loop turtleState = async {
            // read a command message from teh queue
            let! command = inbox.Receive()
            // create a new state from handling the message
            let newState =
                match command with
                | Move distance ->
                    move distance turtleState
                | Turn angle ->
                    turn angle turtleState
                | PenUp ->
                    penUp turtleState
                | PenDown ->
                    penDown turtleState
                | SetColor color ->
                    setColor color turtleState
            return! loop newState
            }
        loop Turtle.initialTurtleState )

    // expose the queue externally
    member this.Post(command) =
        mailboxProc.Post command
```

Sending a command to the Agent

The API calls the agent by constructing a `TurtleCommand` and posting it to the agent's queue.

This time, rather than using the previous approach of "lifting" the `move` command:

```
let stateR = returnR state
let distanceR = validateDistance distanceStr
lift2R move distanceR stateR
```

we'll use the `result` computation expression instead, so the code above would have looked like this:

```
result {
    let! distance = validateDistance distanceStr
    move distance state
    }
```

In the agent implementation, we are not calling a `move` command, but instead creating the `Move` case of the `Command` type, so the code looks like:

```
result {
    let! distance = validateDistance distanceStr
    let command = Move distance
    turtleAgent.Post command
    }
```

Here's the complete code:

```
member this.Exec (commandStr:string) =
    let tokens = commandStr.Split(' ') |> List.ofArray |> List.map trimString

    // calculate the new state
    let result =
        match tokens with
        | [ "Move"; distanceStr ] -> result {
            let! distance = validateDistance distanceStr
            let command = Move distance
            turtleAgent.Post command
            }

        | [ "Turn"; angleStr ] -> result {
            let! angle = validateAngle angleStr
            let command = Turn angle
            turtleAgent.Post command
            }

        | [ "Pen"; "Up" ] -> result {
            let command = PenUp
            turtleAgent.Post command
            }

        | [ "Pen"; "Down" ] -> result {
            let command = PenDown
            turtleAgent.Post command
            }

        | [ "SetColor"; colorStr ] -> result {
            let! color = validateColor colorStr
            let command = SetColor color
            turtleAgent.Post command
            }

        | _ ->
            Failure (InvalidCommand commandStr)

    // return any errors
    result
```

Advantages and disadvantages of the Agent approach

Advantages

- A great way to protect mutable state without using locks.
- The API is decoupled from a particular implementation via the message queue. The `TurtleCommand` acts as a sort of protocol that decouples the two ends of the queue.
- The turtle agent is naturally asynchronous.
- Agents can easily be scaled horizontally.

Disadvantages

- Agents are stateful and have the same problem as stateful objects:
    - It is harder to reason about your code.
    - Testing is harder.
    - It is all too easy to create a web of complex dependencies between actors.
- A robust implementation for agents can get quite complex, as you may need support for supervisors, heartbeats, back pressure, etc.

The source code for this version is available here .

6: Dependency injection using interfaces

All the implementations so far have been tied to a specific implementation of the turtle functions, with the exception of the Agent version, where the API communicated indirectly via a queue.

So let's now look at some ways of decoupling the API from the implementation.

Designing an interface, object-oriented style

We'll start with the classic OO way of decoupling implementations: using interfaces.

Applying that approach to the turtle domain, we can see that our API layer will need to communicate with a `ITurtle` interface rather than a specific turtle implementation. The client injects the turtle implementation later, via the API's constructor.

Here's the interface definition:

```
type ITurtle =
    abstract Move : Distance -> unit
    abstract Turn : Angle -> unit
    abstract PenUp : unit -> unit
    abstract PenDown : unit -> unit
    abstract SetColor : PenColor -> unit
```

Note that there are a lot of `unit` s in these functions. A `unit` in a function signature implies side effects, and indeed the `TurtleState` is not used anywhere, as this is a OO-based approach where the mutable state is encapsulated in the object.

Next, we need to change the API layer to use the interface by injecting it in the constructor for `TurtleApi`. Other than that, the rest of the API code is unchanged, as shown by the snippet below:

```
type TurtleApi(turtle: ITurtle) =

    // other code

    member this.Exec (commandStr:string) =
        let tokens = commandStr.Split(' ') |> List.ofArray |> List.map trimString
        match tokens with
        | [ "Move"; distanceStr ] ->
            let distance = validateDistance distanceStr
            turtle.Move distance
        | [ "Turn"; angleStr ] ->
            let angle = validateAngle angleStr
            turtle.Turn angle
        // etc
```

Creating some implementations of an OO interface

Now let's create and test some implementations.

The first implementation will be called `normalSize` and will be the original one. The second will be called `halfSize` and will reduce all the distances by half.

For `normalSize` we could go back and retrofit the orginal `Turtle` class to support the `ITurtle` interface. But I hate having to change working code! Instead, we can create a "proxy" wrapper around the orginal `Turtle` class, where the proxy implements the new interface.

In some languages, creating proxy wrappers can be long-winded, but in F# you can use object expressions to implement an interface quickly:

```
let normalSize() =
    let log = printfn "%s"
    let turtle = Turtle(log)

    // return an interface wrapped around the Turtle
    {new ITurtle with
        member this.Move dist = turtle.Move dist
        member this.Turn angle = turtle.Turn angle
        member this.PenUp() = turtle.PenUp()
        member this.PenDown() = turtle.PenDown()
        member this.SetColor color = turtle.SetColor color
    }
```

And to create the `halfSize` version, we do the same thing, but intercept the calls to `Move` and halve the distance parameter:

```
let halfSize() =
    let normalSize = normalSize()

    // return a decorated interface
    {new ITurtle with
        member this.Move dist = normalSize.Move (dist/2.0)   // halved!!
        member this.Turn angle = normalSize.Turn angle
        member this.PenUp() = normalSize.PenUp()
        member this.PenDown() = normalSize.PenDown()
        member this.SetColor color = normalSize.SetColor color
    }
```

This is actually the "decorator" pattern at work: we're wrapping `normalSize` in a proxy with an identical interface, then changing the behavior for some of the methods, while passing others though untouched.

Injecting dependencies, OO style

Now let's look at the client code that injects the dependencies into the API.

First, some code to draw a triangle, where a `TurtleApi` is passed in:

```
let drawTriangle(api:TurtleApi) =
    api.Exec "Move 100"
    api.Exec "Turn 120"
    api.Exec "Move 100"
    api.Exec "Turn 120"
    api.Exec "Move 100"
    api.Exec "Turn 120"
```

And now let's try drawing the triangle by instantiating the API object with the normal interface:

```
let iTurtle = normalSize()   // an ITurtle type
let api = TurtleApi(iTurtle)
drawTriangle(api)
```

Obviously, in a real system, the dependency injection would occur away from the call site, using an IoC container or similar.

If we run it, the output of `drawTriangle` is just as before:

```
Move 100.0
...Draw line from (0.0,0.0) to (100.0,0.0) using Black
Turn 120.0
Move 100.0
...Draw line from (100.0,0.0) to (50.0,86.6) using Black
Turn 120.0
Move 100.0
...Draw line from (50.0,86.6) to (0.0,0.0) using Black
Turn 120.0
```

And now with the half-size interface..

```
let iTurtle = halfSize()
let api = TurtleApi(iTurtle)
drawTriangle(api)
```

...the output is, as we hoped, half the size!

```
Move 50.0
...Draw line from (0.0,0.0) to (50.0,0.0) using Black
Turn 120.0
Move 50.0
...Draw line from (50.0,0.0) to (25.0,43.3) using Black
Turn 120.0
Move 50.0
...Draw line from (25.0,43.3) to (0.0,0.0) using Black
Turn 120.0
```

Designing an interface, functional style

In a pure FP world, OO-style interfaces do not exist. However, you can emulate them by using a record containing functions, with one function for each method in the interface.

So let's create a alternative version of dependency injection, where this time the API layer will use a record of functions rather than an interface.

A record of functions is a normal record, but the types of the fields are function types. Here's the definition we'll use:

```
type TurtleFunctions = {
    move : Distance -> TurtleState -> TurtleState
    turn : Angle -> TurtleState -> TurtleState
    penUp : TurtleState -> TurtleState
    penDown : TurtleState -> TurtleState
    setColor : PenColor -> TurtleState -> TurtleState
    }
```

Note that there are no `unit` s in these function signatures, unlike the OO version. Instead, the `TurtleState` is explicitly passed in and returned.

Also note that there is no logging either. The logging method will be baked in to the functions when the record is created.

The `TurtleApi` constructor now takes a `TurtleFunctions` record rather than an `ITurtle` , but as these functions are pure, the API needs to manage the state again with a `mutable` field.

```
type TurtleApi(turtleFunctions: TurtleFunctions) =

    let mutable state = initialTurtleState
```

The implementation of the main `Exec` method is very similar to what we have seen before, with these differences:

- The function is fetched from the record (e.g. `turtleFunctions.move`).
- All the activity takes place in a `result` computation expression so that the result of the validations can be used.

Here's the code:

```
member this.Exec (commandStr:string) =
    let tokens = commandStr.Split(' ') |> List.ofArray |> List.map trimString

    // return Success of unit, or Failure
    match tokens with
    | [ "Move"; distanceStr ] -> result {
        let! distance = validateDistance distanceStr
        let newState = turtleFunctions.move distance state
        updateState newState
        }
    | [ "Turn"; angleStr ] -> result {
        let! angle = validateAngle angleStr
        let newState = turtleFunctions.turn angle state
        updateState newState
        }
    // etc
```

Creating some implementations of a "record of functions"

Noe let's create some implementations.

Again, we'll have a `normalSize` implementation and a `halfSize` implementation.

For `normalSize` we just need to use the functions from the original `Turtle` module, with the logging baked in using partial application:

```
let normalSize() =
    let log = printfn "%s"
    // return a record of functions
    {
        move = Turtle.move log
        turn = Turtle.turn log
        penUp = Turtle.penUp log
        penDown = Turtle.penDown log
        setColor = Turtle.setColor log
    }
```

And to create the `halfSize` version, we clone the record, and change just the `move` function:

```
let halfSize() =
    let normalSize = normalSize()
    // return a reduced turtle
    { normalSize with
        move = fun dist -> normalSize.move (dist/2.0)
    }
```

What's nice about cloning records rather than proxying interfaces is that we don't have to reimplement every function in the record, just the ones we care about.

Injecting dependencies again

The client code that injects the dependencies into the API is implemented just as you expect. The API is a class with a constructor, and so the record of functions can be passed into the constructor in exactly the same way that the `ITurtle` interface was:

```
let turtleFns = normalSize()  // a TurtleFunctions type
let api = TurtleApi(turtleFns)
drawTriangle(api)
```

As you can see, the client code in the `ITurtle` version and `TurtleFunctions` version looks identical! If it wasn't for the different types, you could not tell them apart.

Advantages and disadvantages of using interfaces

The OO-style interface and the FP-style "record of functions" are very similar, although the FP functions are stateless, unlike the OO interface.

Advantages

- The API is decoupled from a particular implementation via the interface.
- For the FP "record of functions" approach (compared to OO interfaces):
    - Records of functions can be cloned more easily than interfaces.
    - The functions are stateless

Disadvantages

- Interfaces are more monolithic than individual functions and can easily grow to include too many unrelated methods, breaking the Interface Segregation Principle if care is not taken.
- Interfaces are not composable (unlike individual functions).
- For more on the problems with this approach, see this Stack Overflow answer by Mark Seemann.
- For the OO interface approach in particular:
    - You may have to modify existing classes when refactoring to an interface.
- For the FP "record of functions" approach:
    - Less tooling support, and poor interop, compared to OO interfaces.

The source code for these versions is available here (interface) and here (record of functions).

7: Dependency injection using functions

The two main disadvantages of the "interface" approach is that interfaces are not composable, and they break the "pass in only the dependencies you need" rule, which is a key part of functional design.

In a true functional approach, we would pass in functions. That is, the API layer communicates via one or more functions that are passed in as parameters to the API call. These functions are typically partially applied so that the call site is decoupled from the "injection".

No interface is passed to the constructor as generally there is no constructor! (I'm only using a API class here to wrap the mutable turtle state.)

In the approach in this section, I'll show two alternatives which use function passing to inject dependencies:

- In the first approach, each dependency (turtle function) is passed separately.
- In the second approach, only one function is passed in. So to determine which specific turtle function is used, a discriminated union type is defined.

Approach 1 - passing in each dependency as a separate function

The simplest way to manage dependencies is always just to pass in all dependencies as parameters to the function that needs them.

In our case, the `Exec` method is the only function that needs to control the turtle, so we can pass them in there directly:

```
member this.Exec move turn penUp penDown setColor (commandStr:string) =
    ...
```

To stress that point again: in this approach dependencies are always passed "just in time", to the function that needs them. No dependencies are used in the constructor and then used later.

Here's a bigger snippet of the `Exec` method using those functions:

```
member this.Exec move turn penUp penDown setColor (commandStr:string) =
    ...

    // return Success of unit, or Failure
    match tokens with
    | [ "Move"; distanceStr ] -> result {
        let! distance = validateDistance distanceStr
        let newState = move distance state   // use `move` function that was passed in
        updateState newState
        }
    | [ "Turn"; angleStr ] -> result {
        let! angle = validateAngle angleStr
        let newState = turn angle state   // use `turn` function that was passed in
        updateState newState
        }
    ...
```

Using partial application to bake in an implementation

To create a normal or half-size version of `Exec` , we just pass in different functions:

```
let log = printfn "%s"
let move = Turtle.move log
let turn = Turtle.turn log
let penUp = Turtle.penUp log
let penDown = Turtle.penDown log
let setColor = Turtle.setColor log

let normalSize() =
    let api = TurtleApi()
    // partially apply the functions
    api.Exec move turn penUp penDown setColor
    // the return value is a function:
    //     string -> Result<unit,ErrorMessage>

let halfSize() =
    let moveHalf dist = move (dist/2.0)
    let api = TurtleApi()
    // partially apply the functions
    api.Exec moveHalf turn penUp penDown setColor
    // the return value is a function:
    //     string -> Result<unit,ErrorMessage>
```

In both cases we are returning a *function* of type `string -> Result<unit,ErrorMessage>` .

Using a purely functional API

So now when we want to draw something, we need only pass in *any* function of type `string -> Result<unit,ErrorMessage>`. The `TurtleApi` is no longer needed or mentioned!

```
// the API type is just a function
type ApiFunction = string -> Result<unit,ErrorMessage>

let drawTriangle(api:ApiFunction) =
    result {
        do! api "Move 100"
        do! api "Turn 120"
        do! api "Move 100"
        do! api "Turn 120"
        do! api "Move 100"
        do! api "Turn 120"
        }
```

And here is how the API would be used:

```
let apiFn = normalSize()  // string -> Result<unit,ErrorMessage>
drawTriangle(apiFn)

let apiFn = halfSize()
drawTriangle(apiFn)
```

So, although we did have mutable state in the `TurtleApi`, the final "published" api is a function that hides that fact.

This approach of having the api be a single function makes it very easy to mock for testing!

```
let mockApi s =
    printfn "[MockAPI] %s" s
    Success ()

drawTriangle(mockApi)
```

Approach 2 - passing a single function that handles all commands

In the version above, we passed in 5 separate functions!

Generally, when you are passing in more than three or four parameters, that implies that your design needs tweaking. You shouldn't really need that many, if the functions are truly independent.

But in our case, the five functions are *not* independent -- they come as a set -- so how can we pass them in together without using a "record of functions" approach?

The trick is to pass in just *one* function! But how can one function handle five different actions? Easy - by using a discriminated union to represent the possible commands.

We've seen this done before in the agent example, so let's revisit that type again:

```
type TurtleCommand =
    | Move of Distance
    | Turn of Angle
    | PenUp
    | PenDown
    | SetColor of PenColor
```

All we need now is a function that handles each case of that type.

Befor we do that though, let's look at the changes to the `Exec` method implementation:

```
member this.Exec turtleFn (commandStr:string) =
    ...

    // return Success of unit, or Failure
    match tokens with
    | [ "Move"; distanceStr ] -> result {
        let! distance = validateDistance distanceStr
        let command =  Move distance      // create a Command object
        let newState = turtleFn command state
        updateState newState
        }
    | [ "Turn"; angleStr ] -> result {
        let! angle = validateAngle angleStr
        let command =  Turn angle      // create a Command object
        let newState = turtleFn command state
        updateState newState
        }
    ...
```

Note that a `command` object is being created and then the `turtleFn` parameter is being called with it.

And by the way, this code is very similar to the agent implementation, which used `turtleAgent.Post command` rather than `newState = turtleFn command state`:

Using partial application to bake in an implementation

Let's create the two implementations using this approach:

```
let log = printfn "%s"
let move = Turtle.move log
let turn = Turtle.turn log
let penUp = Turtle.penUp log
let penDown = Turtle.penDown log
let setColor = Turtle.setColor log

let normalSize() =
    let turtleFn = function
        | Move dist -> move dist
        | Turn angle -> turn angle
        | PenUp -> penUp
        | PenDown -> penDown
        | SetColor color -> setColor color

    // partially apply the function to the API
    let api = TurtleApi()
    api.Exec turtleFn
    // the return value is a function:
    //     string -> Result<unit,ErrorMessage>

let halfSize() =
    let turtleFn = function
        | Move dist -> move (dist/2.0)
        | Turn angle -> turn angle
        | PenUp -> penUp
        | PenDown -> penDown
        | SetColor color -> setColor color

    // partially apply the function to the API
    let api = TurtleApi()
    api.Exec turtleFn
    // the return value is a function:
    //     string -> Result<unit,ErrorMessage>
```

As before, in both cases we are returning a function of type `string -> Result<unit,ErrorMessage>` ,. which we can pass into the `drawTriangle` function we defined earlier:

```
let api = normalSize()
drawTriangle(api)

let api = halfSize()
drawTriangle(api)
```

Advantages and disadvantages of using functions

Advantages

- The API is decoupled from a particular implementation via parameterization.
- Because dependencies are passed in at the point of use ("in your face") rather than in a constructor ("out of sight"), the tendency for dependencies to multiply is greatly reduced.
- Any function parameter is automatically a "one method interface" so no retrofitting is needed.
- Regular partial application can be used to bake in parameters for "dependency injection". No special pattern or IoC container is needed.

Disadvantages

- If the number of dependent functions is too great (say more than four) passing them all in as separate parameters can become awkward (hence, the second approach).
- The discriminated union type can be trickier to work with than an interface.

The source code for these versions is available here (five function params) and here (one function param).

8: Batch processing using a state monad

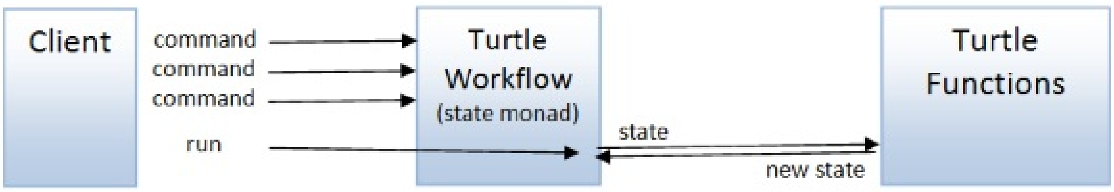
In the next two sections, we'll switch from "interactive" mode, where instructions are processed one at a time, to "batch" mode, where a whole series of instructions are grouped together and then run as one unit.

In the first design, we'll go back to the model where the client uses the Turtle functions directly.

Just as before, the client must keep track of the current state and pass it into the next function call, but this time we'll keep the state out of sight by using a so-called "state monad" to thread the state through the various instructions. As a result, there are no mutables anywhere!

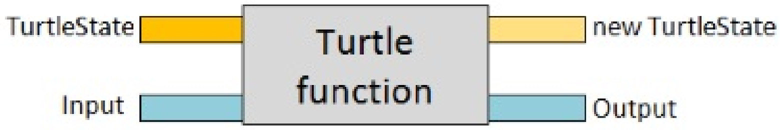
This won't be a generalized state monad, but a simplified one just for this demonstration. I'll call it the `turtle` workflow.

(For more on the state monad see my "monadster" talk and post and post on parser combinators)



Defining the `turtle` workflow

The core turtle functions that we defined at the very beginning follow the same "shape" as many other state-transforming functions, an input plus the turtle state, and the output plus the turtle state.

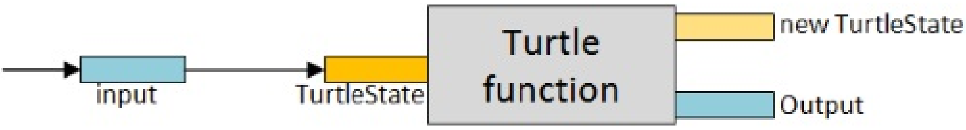


(It's true that, so far. we have not had any useable output from the turtle functions, but in a later example we will see this output being used to make decisions.)

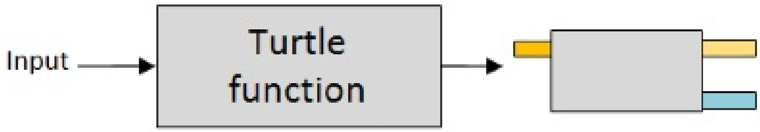
There is a standard way to deal with these kinds of functions -- the "state monad".

Let's look at how this is built.

First, note that, thanks to currying, we can recast a function in this shape into two separate one-parameter functions: processing the input generates another function that in turn has the state as the parameter:



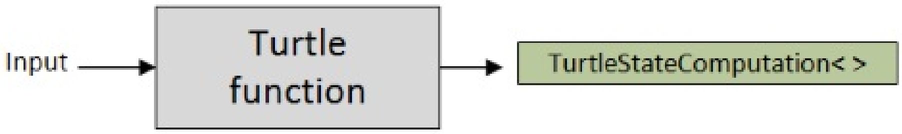
We can then think of a turtle function as something that takes an input and returns a new *function*, like this:



In our case, using `TurtleState` as the state, the returned function will look like this:

```
TurtleState -> 'a * TurtleState
```

Finally, to make it easier to work with, we can treat the returned function as a thing in its own right, give it a name such as `TurtleStateComputation` :



In the implementation, we would typically wrap the function with a single case discriminated union like this:

```
type TurtleStateComputation<'a> =
    TurtleStateComputation of (Turtle.TurtleState -> 'a * Turtle.TurtleState)
```

So that is the basic idea behind the "state monad". However, it's important to realize that a state monad consists of more than just this type -- you also need some functions ("return" and "bind") that obey some sensible laws.

I won't define the `returnT` and `bindT` functions here, but you can see their definitions in the full source.

We need some additional helper functions too. (I'm going to add a `T` for Turtle suffix to all the functions).

In particular, we need a way to feed some state into the `TurtleStateComputation` to "run" it:

```
let runT turtle state =
    // pattern match against the turtle
    // to extract the inner function
    let (TurtleStateComputation innerFn) = turtle
    // run the inner function with the passed in state
    innerFn state
```

Finally, we can create a `turtle` workflow, which is a computation expression that makes it easier to work with the `TurtleStateComputation` type:

```
// define a computation expression builder
type TurtleBuilder() =
    member this.Return(x) = returnT x
    member this.Bind(x,f) = bindT f x

// create an instance of the computation expression builder
let turtle = TurtleBuilder()
```

Using the Turtle workflow

To use the `turtle` workflow, we first need to create "lifted" or "monadic" versions of the turtle functions:

```
let move dist =
    toUnitComputation (Turtle.move log dist)
// val move : Distance -> TurtleStateComputation<unit>

let turn angle =
    toUnitComputation (Turtle.turn log angle)
// val turn : Angle -> TurtleStateComputation<unit>

let penDown =
    toUnitComputation (Turtle.penDown log)
// val penDown : TurtleStateComputation<unit>

let penUp =
    toUnitComputation (Turtle.penUp log)
// val penUp : TurtleStateComputation<unit>

let setColor color =
    toUnitComputation (Turtle.setColor log color)
// val setColor : PenColor -> TurtleStateComputation<unit>
```

The `toUnitComputation` helper function does the lifting. Don't worry about how it works, but the effect is that the original version of the `move` function (`Distance -> TurtleState -> TurtleState`) is reborn as a function returning a `TurtleStateComputation` (`Distance -> TurtleStateComputation<unit>`)

Once we have these "monadic" versions, we can use them inside the `turtle` workflow like this:

```
let drawTriangle() =
    // define a set of instructions
    let t = turtle {
        do! move 100.0
        do! turn 120.0<Degrees>
        do! move 100.0
        do! turn 120.0<Degrees>
        do! move 100.0
        do! turn 120.0<Degrees>
        }

    // finally, run them using the initial state as input
    runT t initialTurtleState
```

The first part of `drawTriangle` chains together six instructions, but importantly, does *not* run them. Only when the `runT` function is used at the end are the instructions actually executed.

The `drawPolygon` example is a little more complicated. First we define a workflow for drawing one side:

```
let oneSide = turtle {
    do! move 100.0
    do! turn angleDegrees
    }
```

But then we need a way of combining all the sides into a single workflow. There are a couple of ways of doing this. I'll go with creating a pairwise combiner `chain` and then using `reduce` to combine all the sides into one operation.

```
// chain two turtle operations in sequence
let chain f g  = turtle {
    do! f
    do! g
    }

// create a list of operations, one for each side
let sides = List.replicate n oneSide

// chain all the sides into one operation
let all = sides |> List.reduce chain
```

Here's the complete code for `drawPolygon` :

```
let drawPolygon n =
    let angle = 180.0 - (360.0/float n)
    let angleDegrees = angle * 1.0<Degrees>

    // define a function that draws one side
    let oneSide = turtle {
        do! move 100.0
        do! turn angleDegrees
        }

    // chain two turtle operations in sequence
    let chain f g  = turtle {
        do! f
        do! g
        }

    // create a list of operations, one for each side
    let sides = List.replicate n oneSide

    // chain all the sides into one operation
    let all = sides |> List.reduce chain

    // finally, run them using the initial state
    runT all initialTurtleState
```

Advantages and disadvantages of the `turtle` workflow

Advantages

- The client code is similar to imperative code, but preserves immutability.
- The workflows are composable -- you can define two workflows and then combine them to create another workflow.

Disadvantages

- Coupled to a particular implementation of the turtle functions.
- More complex than tracking state explicitly.
- Stacks of nested monads/workflows are hard to work with.

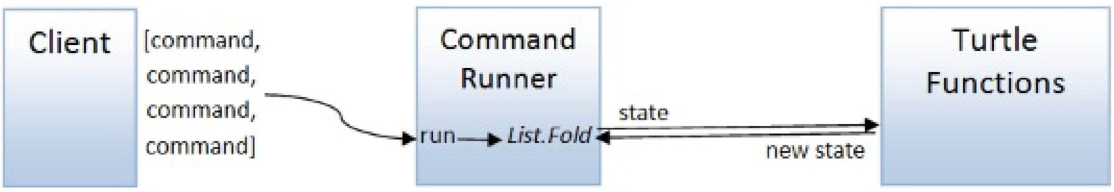
As an example of that last point, let's say we have a `seq` containing a `result` workflow containing a `turtle` workflow and we want to invert them so that the `turtle` workflow is on the outside. How would you do that? It's not obvious!

The source code for this version is available here.

9: Batch processing using command objects

Another batch-oriented approach is to reuse the `TurtleCommand` type in a new way. Instead of calling functions immediately, the client creates a list of commands that will be run as a group.

When you "run" the list of commands, you can just execute each one in turn using the standard Turtle library functions, using `fold` to thread the state through the sequence.



And since all the commands are run at once, this approach means that there is no state that needs to be persisted between calls by the client.

Here's the `TurtleCommand` definition again:

```
type TurtleCommand =
    | Move of Distance
    | Turn of Angle
    | PenUp
    | PenDown
    | SetColor of PenColor
```

To process a sequence of commands, we will need to fold over them, threading the state through, so we need a function that applies a single command to a state and returns a new state:

```
/// Apply a command to the turtle state and return the new state
let applyCommand state command =
    match command with
    | Move distance ->
        move distance state
    | Turn angle ->
        turn angle state
    | PenUp ->
        penUp state
    | PenDown ->
        penDown state
    | SetColor color ->
        setColor color state
```

And then, to run all the commands, we just use `fold`:

```
/// Run list of commands in one go
let run aListOfCommands =
    aListOfCommands
    |> List.fold applyCommand Turtle.initialTurtleState
```

Running a batch of Commands

To draw a triangle, say, we just create a list of the commands and then run them:

```
let drawTriangle() =
    // create the list of commands
    let commands = [
        Move 100.0
        Turn 120.0<Degrees>
        Move 100.0
        Turn 120.0<Degrees>
        Move 100.0
        Turn 120.0<Degrees>
        ]
    // run them
    run commands
```

Now, since the commands are just a collection, we can easily build bigger collections from smaller ones.

Here's an example for `drawPolygon`, where `drawOneSide` returns a collection of commands, and that collection is duplicated for each side:

```
let drawPolygon n =
    let angle = 180.0 - (360.0/float n)
    let angleDegrees = angle * 1.0<Degrees>

    // define a function that draws one side
    let drawOneSide sideNumber = [
        Move 100.0
        Turn angleDegrees
        ]

    // repeat for all sides
    let commands =
        [1..n] |> List.collect drawOneSide

    // run the commands
    run commands
```

Advantages and disadvantages of batch commands

Advantages

- Simpler to construct and use than workflows or monads.
- Only one function is coupled to a particular implementation. The rest of the client is decoupled.

Disadvantages

- Batch oriented only.

- Only suitable when control flow is *not* based on the response from a previous command. If you *do* need to respond to the result of each command, consider using the "interpreter" approach discussed later.

The source code for this version is available here.

Interlude: Conscious decoupling with data types

In three of the examples so far (the agent, functional dependency injection and batch processing) we have used a `Command` type -- a discriminated union containing a case for each API call. We'll also see something similar used for the event sourcing and interpreter approaches in the next post.

This is not an accident. One of the differences between object-oriented design and functional design is that OO design focuses on behavior, while functional design focuses on data transformation.

As a result, their approach to decoupling differs too. OO designs prefer to provide decoupling by sharing bundles of encapsulated behavior ("interfaces") while functional designs prefer to provide decoupling by agreeing on a common data type, sometimes called a "protocol" (although I prefer to reserve that word for message exchange patterns).

Once that common data type is agreed upon, any function that emits that type can be connected to any function that consumes that type using regular function composition.

You can also think of the two approaches as analogous to the choice between RPC or message-oriented APIs in web services, and just as message-based designs have many advantages over RPC, so the data-based decoupling has similar advantages over the behavior-based decoupling.

Some advantages of decoupling using data include:

- Using a shared data type means that composition is trivial. It is harder to compose behavior-based interfaces.
- *Every* function is already "decoupled", as it were, and so there is no need to retrofit existing functions when refactoring. At worst you might need to convert one data type to another, but that is easily accomplished using... moar functions and moar function composition!
- Data structures are easy to serialize to remote services if and when you need to split your code into physically separate services.

- Data structures are easy to evolve safely. For example, if I added a sixth turtle action, or removed an action, or changed the parameters of an action, the discriminated union type would change and all clients of the shared type would fail to compile until the sixth turtle action is accounted for, etc. On the other hand, if you *didn't* want existing code to break, you can use a versioning-friendly data serialization format like protobuf. Neither of these options are as easy when interfaces are used.

Summary

> The meme is spreading.
> The turtle must be paddling.
> -- *"Thirteen ways of looking at a turtle", by Wallace D Coriacea*

Hello? Anyone still there? Thanks for making it this far!

So, time for a break! In the next post, we'll cover the remaining four ways of looking at a turtle.

The source code for this post is available on github.

Thirteen ways of looking at a turtle (part 2)

> This post is part of the F# Advent Calendar in English 2015 project. Check out all the other great posts there! And special thanks to Sergey Tihon for organizing this.

In this two-part mega-post, I'm stretching the simple turtle graphics model to the limit while demonstrating partial application, validation, the concept of "lifting", agents with message queues, dependency injection, the State monad, event sourcing, stream processing, and an interpreter!

In the previous post, we covered the first nine ways of looking at a turtle. In this post, we'll look at the remaining four.

As a reminder, here are the thirteen ways:

- Way 1. A basic object-oriented approach, in which we create a class with mutable state.
- Way 2. A basic functional approach, in which we create a module of functions with immutable state.
- Way 3. An API with a object-oriented core, in which we create an object-oriented API that calls a stateful core class.
- Way 4. An API with a functional core, in which we create an stateful API that uses stateless core functions.
- Way 5. An API in front of an agent, in which we create an API that uses a message queue to communicate with an agent.
- Way 6. Dependency injection using interfaces, in which we decouple the implementation from the API using an interface or record of functions.
- Way 7. Dependency injection using functions, in which we decouple the implementation from the API by passing a function parameter.
- Way 8. Batch processing using a state monad, in which we create a special "turtle workflow" computation expression to track state for us.
- Way 9. Batch processing using command objects, in which we create a type to represent a turtle command, and then process a list of commands all at once.
- Interlude: Conscious decoupling with data types. A few notes on using data vs. interfaces for decoupling.
- Way 10. Event sourcing, in which state is built from a list of past events.
- Way 11. Functional Retroactive Programming (stream processing), in which business logic is based on reacting to earlier events.
- Episode V: The Turtle Strikes Back, in which the turtle API changes so that some commands may fail.
- Way 12. Monadic control flow, in which we make decisions in the turtle workflow based

on results from earlier commands.
- Way 13. A turtle interpreter, in which we completely decouple turtle programming from turtle implementation, and nearly encounter the free monad.
- Review of all the techniques used.

and 2 bonus ways for the extended edition:

- Way 14. Abstract Data Turtle, in which we encapsulate the details of a turtle implementation by using an Abstract Data Type.
- Way 15. Capability-based Turtle, in which we control what turtle functions are available to a client, based on the current state of the turtle.

It's turtles all the way down!

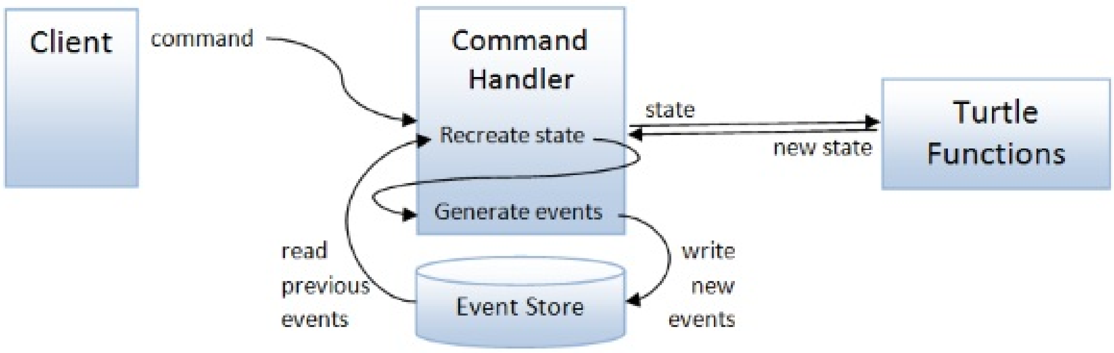
All source code for this post is available on github.

10: Event sourcing -- Building state from a list of past events

In this design, we build on the "command" concept used in the Agent (way 5) and Batch (way 9) approaches, but replacing "commands" with "events" as the method of updating state.

The way that it works is:

- The client sends a `Command` to a `CommandHandler` .
- Before processing a `Command` , the `CommandHandler` first rebuilds the current state from scratch using the past events associated with that particular turtle.
- The `CommandHandler` then validates the command and decides what to do based on the current (rebuilt) state. It generates a (possibly empty) list of events.
- The generated events are stored in an `EventStore` for the next command to use.



In this way, neither the client nor the command handler needs to track state. Only the `EventStore` is mutable.

The Command and Event types

We will start by defining the types relating to our event sourcing system. First, the types related to commands:

```
type TurtleId = System.Guid

/// A desired action on a turtle
type TurtleCommandAction =
    | Move of Distance
    | Turn of Angle
    | PenUp
    | PenDown
    | SetColor of PenColor

/// A command representing a desired action addressed to a specific turtle
type TurtleCommand = {
    turtleId : TurtleId
    action : TurtleCommandAction
    }
```

Note that the command is addressed to a particular turtle using a `TurtleId`.

Next, we will define two kinds of events that can be generated from a command:

- A `StateChangedEvent` which represents what changed in the state
- A `MovedEvent` which represents the start and end positions of a turtle movement.

```
/// An event representing a state change that happened
type StateChangedEvent =
    | Moved of Distance
    | Turned of Angle
    | PenWentUp
    | PenWentDown
    | ColorChanged of PenColor

/// An event representing a move that happened
/// This can be easily translated into a line-drawing activity on a canvas
type MovedEvent = {
    startPos : Position
    endPos : Position
    penColor : PenColor option
    }

/// A union of all possible events
type TurtleEvent =
    | StateChangedEvent of StateChangedEvent
    | MovedEvent of MovedEvent
```

It is an important part of event sourcing that all events are labeled in the past tense: `Moved` and `Turned` rather than `Move` and `Turn`. The event are facts -- they have happened in the past.

The Command handler

The next step is to define the functions that convert a command into events.

We will need:

- A (private) `applyEvent` function that updates the state from a previous event.
- A (private) `eventsFromCommand` function that determines what events to generate, based on the command and the state.
- A public `commandHandler` function that handles the command, reads the events from the event store and calls the other two functions.

Here's `applyEvent`. You can see that it is very similar to the `applyCommand` function that we saw in the previous batch-processing example.

```
/// Apply an event to the current state and return the new state of the turtle
let applyEvent log oldState event =
    match event with
    | Moved distance ->
        Turtle.move log distance oldState
    | Turned angle ->
        Turtle.turn log angle oldState
    | PenWentUp ->
        Turtle.penUp log oldState
    | PenWentDown ->
        Turtle.penDown log oldState
    | ColorChanged color ->
        Turtle.setColor log color oldState
```

The `eventsFromCommand` function contains the key logic for validating the command and creating events.

- In this particular design, the command is always valid, so at least one event is returned.
- The `StateChangedEvent` is created from the `TurtleCommand` in a direct one-to-one map of the cases.
- The `MovedEvent` is only created from the `TurtleCommand` if the turtle has changed position.

```
// Determine what events to generate, based on the command and the state.
let eventsFromCommand log command stateBeforeCommand =

    // --------------------------
    // create the StateChangedEvent from the TurtleCommand
    let stateChangedEvent =
        match command.action with
        | Move dist -> Moved dist
        | Turn angle -> Turned angle
        | PenUp -> PenWentUp
        | PenDown -> PenWentDown
        | SetColor color -> ColorChanged color

    // --------------------------
    // calculate the current state from the new event
    let stateAfterCommand =
        applyEvent log stateBeforeCommand stateChangedEvent

    // --------------------------
    // create the MovedEvent
    let startPos = stateBeforeCommand.position
    let endPos = stateAfterCommand.position
    let penColor =
        if stateBeforeCommand.penState=Down then
            Some stateBeforeCommand.color
        else
            None

    let movedEvent = {
        startPos = startPos
        endPos = endPos
        penColor = penColor
        }

    // --------------------------
    // return the list of events
    if startPos <> endPos then
        // if the turtle has moved, return both the stateChangedEvent and the movedEve
nt
        // lifted into the common TurtleEvent type
        [ StateChangedEvent stateChangedEvent; MovedEvent movedEvent]
    else
        // if the turtle has not moved, return just the stateChangedEvent
        [ StateChangedEvent stateChangedEvent]
```

Finally, the `commandHandler` is the public interface. It is passed in some dependencies as parameters: a logging function, a function to retrieve the historical events from the event store, and a function to save the newly generated events into the event store.

```
/// The type representing a function that gets the StateChangedEvents for a turtle id
/// The oldest events are first
type GetStateChangedEventsForId =
     TurtleId -> StateChangedEvent list

/// The type representing a function that saves a TurtleEvent
type SaveTurtleEvent =
    TurtleId -> TurtleEvent -> unit

/// main function : process a command
let commandHandler
    (log:string -> unit)
    (getEvents:GetStateChangedEventsForId)
    (saveEvent:SaveTurtleEvent)
    (command:TurtleCommand) =

    /// First load all the events from the event store
    let eventHistory =
        getEvents command.turtleId

    /// Then, recreate the state before the command
    let stateBeforeCommand =
        let nolog = ignore // no logging when recreating state
        eventHistory
        |> List.fold (applyEvent nolog) Turtle.initialTurtleState

    /// Construct the events from the command and the stateBeforeCommand
    /// Do use the supplied logger for this bit
    let events = eventsFromCommand log command stateBeforeCommand

    // store the events in the event store
    events |> List.iter (saveEvent command.turtleId)
```

Calling the command handler

Now we are ready to send events to the command handler.

First we need some helper functions that create commands:

```
// Command versions of standard actions
let turtleId = System.Guid.NewGuid()
let move dist = {turtleId=turtleId; action=Move dist}
let turn angle = {turtleId=turtleId; action=Turn angle}
let penDown = {turtleId=turtleId; action=PenDown}
let penUp = {turtleId=turtleId; action=PenUp}
let setColor color = {turtleId=turtleId; action=SetColor color}
```

And then we can draw a figure by sending the various commands to the command handler:

```
let drawTriangle() =
    let handler = makeCommandHandler()
    handler (move 100.0)
    handler (turn 120.0<Degrees>)
    handler (move 100.0)
    handler (turn 120.0<Degrees>)
    handler (move 100.0)
    handler (turn 120.0<Degrees>)
```

NOTE: I have not shown how to create the command handler or event store, see the code for full details.

Advantages and disadvantages of event sourcing

Advantages

- All code is stateless, hence easy to test.
- Supports replay of events.

Disadvantages

- Can be more complex to implement than a CRUD approach (or at least, less support from tools and libraries).
- If care is not taken, the command handler can get overly complex and evolve into implementing too much business logic.

The source code for this version is available here.

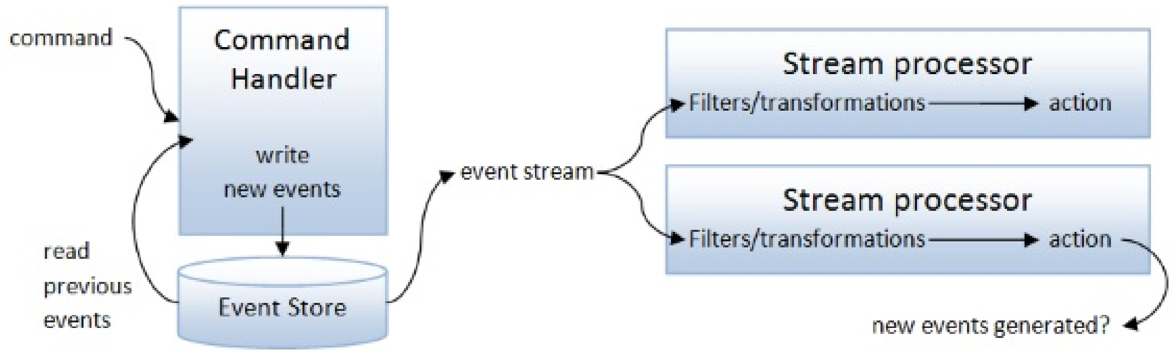
11: Functional Retroactive Programming (stream processing)

In the event sourcing example above, all the domain logic (in our case, just tracing the state) is embedded in the command handler. One drawback of this is that, as the application evolves, the logic in the command handler can become very complex.

A way to avoid this is to combine "functional reactive programming" with event sourcing to create a design where the domain logic is performed on the "read-side", by listening to events ("signals") emitted from the event store.

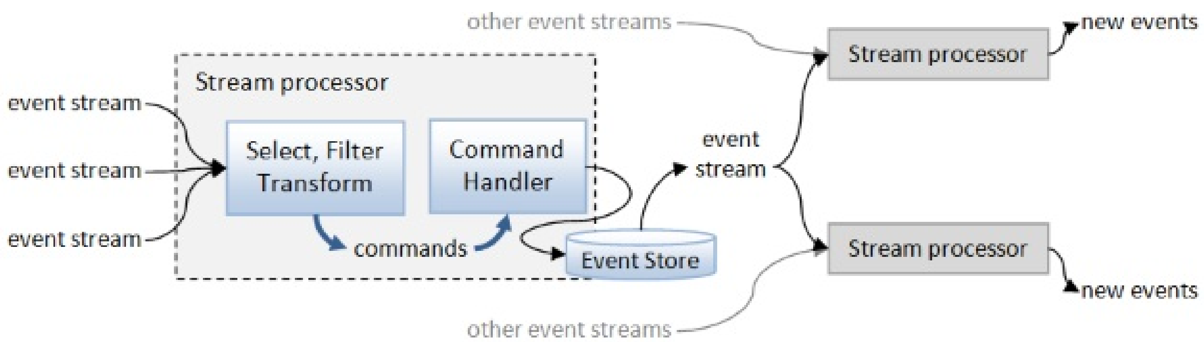
In this approach, the "write-side" follows the same pattern as the event-sourcing example. A client sends a `Command` to a `commandHandler`, which converts that to a list of events and stores them in an `EventStore`.

However the `commandHandler` only does the *minimal* amount of work, such as updating state, and does NOT do any complex domain logic. The complex logic is performed by one or more downstream "processors" (also sometimes called "aggregators") that subscribe to the event stream.



You can even think of these events as "commands" to the processors, and of course, the processors can generate new events for another processor to consume, so this approach can be extended into an architectural style where an application consists of a set of command handlers linked by an event store.

This techique is often called "stream processing". However, Jessica Kerr once called this approach "Functional Retroactive Programming" -- I like that, so I'm going to steal that name!



Implementing the design

For this implementation, the `commandHandler` function is the same as in the event sourcing example, except that no work (just logging!) is done at all. The command handler *only* rebuilds state and generates events. How the events are used for business logic is no longer in its scope.

The new stuff comes in creating the processors.

However, before we can create a processor, we need some helper functions that can filter the event store feed to only include turtle specific events, and of those only `StateChangedEvent` s or `MovedEvent` s.

```
// filter to choose only TurtleEvents
let turtleFilter ev =
    match box ev with
    | :? TurtleEvent as tev -> Some tev
    | _ -> None

// filter to choose only MovedEvents from TurtleEvents
let moveFilter = function
    | MovedEvent ev -> Some ev
    | _ -> None

// filter to choose only StateChangedEvent from TurtleEvents
let stateChangedEventFilter = function
    | StateChangedEvent ev -> Some ev
    | _ -> None
```

Now let's create a processor that listens for movement events and moves a physical turtle when the virtual turtle is moved.

We will make the input to the processor be an `IObservable` -- an event stream -- so that it is not coupled to any specific source such as the `EventStore`. We will connect the `EventStore` "save" event to this processor when the application is configured.

```
/// Physically move the turtle
let physicalTurtleProcessor (eventStream:IObservable<Guid*obj>) =

    // the function that handles the input from the observable
    let subscriberFn (ev:MovedEvent) =
        let colorText =
            match ev.penColor with
            | Some color -> sprintf "line of color %A" color
            | None -> "no line"
        printfn "[turtle  ]: Moved from (%0.2f,%0.2f) to (%0.2f,%0.2f) with %s"
            ev.startPos.x ev.startPos.y ev.endPos.x ev.endPos.y colorText

    // start with all events
    eventStream
    // filter the stream on just TurtleEvents
    |> Observable.choose (function (id,ev) -> turtleFilter ev)
    // filter on just MovedEvents
    |> Observable.choose moveFilter
    // handle these
    |> Observable.subscribe subscriberFn
```

In this case we are just printing the movement -- I'll leave the building of an actual Lego Mindstorms turtle as an exercise for the reader!

Let's also create a processor that draws lines on a graphics display:

```
/// Draw lines on a graphics device
let graphicsProcessor (eventStream:IObservable<Guid*obj>) =

    // the function that handles the input from the observable
    let subscriberFn (ev:MovedEvent) =
        match ev.penColor with
        | Some color ->
            printfn "[graphics]: Draw line from (%0.2f,%0.2f) to (%0.2f,%0.2f) with co
lor %A"
                ev.startPos.x ev.startPos.y ev.endPos.x ev.endPos.y color
        | None ->
            ()  // do nothing

    // start with all events
    eventStream
    // filter the stream on just TurtleEvents
    |> Observable.choose (function (id,ev) -> turtleFilter ev)
    // filter on just MovedEvents
    |> Observable.choose moveFilter
    // handle these
    |> Observable.subscribe subscriberFn
```

And finally, let's create a processor that accumulates the total distance moved so that we can keep track of how much ink has been used, say.

```
/// Listen for "moved" events and aggregate them to keep
/// track of the total ink used
let inkUsedProcessor (eventStream:IObservable<Guid*obj>) =

    // Accumulate the total distance moved so far when a new event happens
    let accumulate distanceSoFar (ev:StateChangedEvent) =
        match ev with
        | Moved dist ->
            distanceSoFar + dist
        | _ ->
            distanceSoFar

    // the function that handles the input from the observable
    let subscriberFn distanceSoFar  =
        printfn "[ink used]: %0.2f" distanceSoFar

    // start with all events
    eventStream
    // filter the stream on just TurtleEvents
    |> Observable.choose (function (id,ev) -> turtleFilter ev)
    // filter on just StateChangedEvent
    |> Observable.choose stateChangedEventFilter
    // accumulate total distance
    |> Observable.scan accumulate 0.0
    // handle these
    |> Observable.subscribe subscriberFn
```

This processor uses `Observable.scan` to accumulate the events into a single value -- the total distance travelled.

Processors in practice

Let's try these out!

For example, here is `drawTriangle` :

```
let drawTriangle() =
    // clear older events
    eventStore.Clear turtleId

    // create an event stream from an IEvent
    let eventStream = eventStore.SaveEvent :> IObservable<Guid*obj>

    // register the processors
    use physicalTurtleProcessor = EventProcessors.physicalTurtleProcessor eventStream
    use graphicsProcessor = EventProcessors.graphicsProcessor eventStream
    use inkUsedProcessor = EventProcessors.inkUsedProcessor eventStream

    let handler = makeCommandHandler
    handler (move 100.0)
    handler (turn 120.0<Degrees>)
    handler (move 100.0)
    handler (turn 120.0<Degrees>)
    handler (move 100.0)
    handler (turn 120.0<Degrees>)
```

Note that `eventStore.SaveEvent` is cast into an `IObservable<Guid*obj>` (that is, an event stream) before being passed to the processors as a parameter.

`drawTriangle` generates this output:

```
[ink used]: 100.00
[turtle  ]: Moved from (0.00,0.00) to (100.00,0.00) with line of color Black
[graphics]: Draw line from (0.00,0.00) to (100.00,0.00) with color Black
[ink used]: 100.00
[ink used]: 200.00
[turtle  ]: Moved from (100.00,0.00) to (50.00,86.60) with line of color Black
[graphics]: Draw line from (100.00,0.00) to (50.00,86.60) with color Black
[ink used]: 200.00
[ink used]: 300.00
[turtle  ]: Moved from (50.00,86.60) to (0.00,0.00) with line of color Black
[graphics]: Draw line from (50.00,86.60) to (0.00,0.00) with color Black
[ink used]: 300.00
```

You can see that all the processors are handling events successfully.

The turtle is moving, the graphics processor is drawing lines, and the ink used processor has correctly calculated the total distance moved as 300 units.

Note, though, that the ink used processor is emitting output on *every* state change (such as turning), rather than only when actual movement happens.

We can fix this by putting a pair `(previousDistance, currentDistance)` in the stream, and then filtering out those events where the values are the same.

Here's the new `inkUsedProcessor` code, with the following changes:

- The `accumulate` function now emits a pair.
- There is a new filter `changedDistanceOnly` .

```
/// Listen for "moved" events and aggregate them to keep
/// track of the total distance moved
/// NEW! No duplicate events!
let inkUsedProcessor (eventStream:IObservable<Guid*obj>) =

    // Accumulate the total distance moved so far when a new event happens
    let accumulate (prevDist,currDist) (ev:StateChangedEvent) =
        let newDist =
            match ev with
            | Moved dist ->
                currDist + dist
            | _ ->
                currDist
        (currDist, newDist)

    // convert unchanged events to None so they can be filtered out with "choose"
    let changedDistanceOnly (currDist, newDist) =
        if currDist <> newDist then
            Some newDist
        else
            None

    // the function that handles the input from the observable
    let subscriberFn distanceSoFar  =
        printfn "[ink used]: %0.2f" distanceSoFar

    // start with all events
    eventStream
    // filter the stream on just TurtleEvents
    |> Observable.choose (function (id,ev) -> turtleFilter ev)
    // filter on just StateChangedEvent
    |> Observable.choose stateChangedEventFilter
    // NEW! accumulate total distance as pairs
    |> Observable.scan accumulate (0.0,0.0)
    // NEW! filter out when distance has not changed
    |> Observable.choose changedDistanceOnly
    // handle these
    |> Observable.subscribe subscriberFn
```

With these changes, the output of `drawTriangle` looks like this:

```
[ink used]: 100.00
[turtle  ]: Moved from (0.00,0.00) to (100.00,0.00) with line of color Black
[graphics]: Draw line from (0.00,0.00) to (100.00,0.00) with color Black
[ink used]: 200.00
[turtle  ]: Moved from (100.00,0.00) to (50.00,86.60) with line of color Black
[graphics]: Draw line from (100.00,0.00) to (50.00,86.60) with color Black
[ink used]: 300.00
[turtle  ]: Moved from (50.00,86.60) to (0.00,0.00) with line of color Black
[graphics]: Draw line from (50.00,86.60) to (0.00,0.00) with color Black
```

and there are no longer any duplicate messages from the `inkUsedProcessor` .

Advantages and disadvantages of stream processing

Advantages

- Same advantages as event-sourcing.
- Decouples stateful logic from other non-intrinsic logic.
- Easy to add and remove domain logic without affecting the core command handler.

Disadvantages

- More complex to implement.

The source code for this version is available here.

Episode V: The Turtle Strikes Back

So far, we have not had to make decisions based on the turtle's state. So, for the two final approaches, we will change the turtle API so that some commands may fail.

For example, we might say that the turtle must move within a limited arena, and a `move` instruction may cause the turtle to hit the barrier. In this case, the `move` instruction can return a choice of `MovedOk` or `HitBarrier` .

Or let's say that there is only a limited amount of colored ink. In this case, trying to set the color may return an "out of ink" response.

So let's update the turtle functions with these cases. First the new response types for `move` and `setColor` :

```
type MoveResponse =
    | MoveOk
    | HitABarrier

type SetColorResponse =
    | ColorOk
    | OutOfInk
```

We will need a bounds checker to see if the turtle is in the arena. Say that if the position tries to go outside the square (0,0,100,100), the response is `HitABarrier` :

```
// if the position is outside the square (0,0,100,100)
// then constrain the position and return HitABarrier
let checkPosition position =
    let isOutOfBounds p =
        p > 100.0 || p < 0.0
    let bringInsideBounds p =
        max (min p 100.0) 0.0

    if isOutOfBounds position.x || isOutOfBounds position.y then
        let newPos = {
            x = bringInsideBounds position.x
            y = bringInsideBounds position.y }
        HitABarrier,newPos
    else
        MoveOk,position
```

And finally, the `move` function needs an extra line to check the new position:

```
let move log distance state =
    let newPosition = ...

    // adjust the new position if out of bounds
    let moveResult, newPosition = checkPosition newPosition

    ...
```

Here's the complete `move` function:

```
let move log distance state =
    log (sprintf "Move %0.1f" distance)
    // calculate new position
    let newPosition = calcNewPosition distance state.angle state.position
    // adjust the new position if out of bounds
    let moveResult, newPosition = checkPosition newPosition
    // draw line if needed
    if state.penState = Down then
        dummyDrawLine log state.position newPosition state.color
    // return the new state and the Move result
    let newState = {state with position = newPosition}
    (moveResult,newState)
```

We will make similar changes for the `setColor` function too, returning `OutOfInk` if we attempt to set the color to `Red`.

```
let setColor log color state =
    let colorResult =
        if color = Red then OutOfInk else ColorOk
    log (sprintf "SetColor %A" color)
    // return the new state and the SetColor result
    let newState = {state with color = color}
    (colorResult,newState)
```

With the new versions of the turtle functions available, we have to create implementations that can respond to the error cases. That will be done in the next two examples.

The source code for the new turtle functions is available here.

12: Monadic control flow

In this approach, we will reuse the `turtle` workflow from way 8. This time though, we will make decisions for the next command based on the result of the previous one.

Before we do that though, let's look at what effect the change to `move` will have on our code. Let's say that we want to move forwards a few times using `move 40.0`, say.

If we write the code using `do!` as we did before, we get a nasty compiler error:

```
let drawShape() =
    // define a set of instructions
    let t = turtle {
        do! move 60.0
        // error FS0001:
        // This expression was expected to have type
        //    Turtle.MoveResponse
        // but here has type
        //     unit
        do! move 60.0
        }
    // etc
```

Instead, we need to use `let!` and assign the response to something.

In the following code, we assign the response to a value and then ignore it!

```
let drawShapeWithoutResponding() =
    // define a set of instructions
    let t = turtle {
        let! response = move 60.0
        let! response = move 60.0
        let! response = move 60.0
        return ()
        }

    // finally, run the monad using the initial state
    runT t initialTurtleState
```

The code does compile and work, but if we run it the output shows that, by the third call, we are banging our turtle against the wall (at 100,0) and not moving anywhere.

```
Move 60.0
...Draw line from (0.0,0.0) to (60.0,0.0) using Black
Move 60.0
...Draw line from (60.0,0.0) to (100.0,0.0) using Black
Move 60.0
...Draw line from (100.0,0.0) to (100.0,0.0) using Black
```

Making decisions based on a response

Let's say that our response to a `move` that returns `HitABarrier` is to turn 90 degrees and wait for the next command. Not the cleverest algorithm, but it will do for demonstration purposes!

Let's design a function to implement this. The input will be a `MoveResponse`, but what will the output be? We want to encode the `turn` action somehow, but the raw `turn` function needs state input that we don't have. So instead let's return a `turtle` workflow that represents the instruction we *want* to do, when the state becomes available (in the `run` command).

So here is the code:

```
let handleMoveResponse moveResponse = turtle {
    match moveResponse with
    | Turtle.MoveOk ->
        () // do nothing
    | Turtle.HitABarrier ->
        // turn 90 before trying again
        printfn "Oops -- hit a barrier -- turning"
        do! turn 90.0<Degrees>
    }
```

The type signature looks like this:

```
val handleMoveResponse : MoveResponse -> TurtleStateComputation<unit>
```

which means that it is a monadic (or "diagonal") function -- one that starts in the normal world and ends in the `TurtleStateComputation` world.

These are exactly the functions that we can use "bind" with, or within computation expressions, `let!` or `do!`.

Now we can add this `handleMoveResponse` step after `move` in the turtle workflow:

```
let drawShape() =
    // define a set of instructions
    let t = turtle {
        let! response = move 60.0
        do! handleMoveResponse response

        let! response = move 60.0
        do! handleMoveResponse response

        let! response = move 60.0
        do! handleMoveResponse response
        }

    // finally, run the monad using the initial state
    runT t initialTurtleState
```

And the result of running it is:

```
Move 60.0
...Draw line from (0.0,0.0) to (60.0,0.0) using Black
Move 60.0
...Draw line from (60.0,0.0) to (100.0,0.0) using Black
Oops -- hit a barrier -- turning
Turn 90.0
Move 60.0
...Draw line from (100.0,0.0) to (100.0,60.0) using Black
```

You can see that the move response worked. When the turtle hit the edge at (100,0) it turned 90 degrees and the next move succeeded (from (100,0) to (100,60)).

So there you go! This code demonstrates how you can make decisions inside the `turtle` workflow while the state is being passed around behind the scenes.

Advantages and disadvantages

Advantages

- Computation expressions allow the code to focus on the logic while taking care of the "plumbing" -- in this case, the turtle state.

Disadvantages

- Still coupled to a particular implementation of the turtle functions.
- Computation expressions can be complex to implement and how they work is not obvious for beginners.

The source code for this version is available here.

13: A turtle interpreter

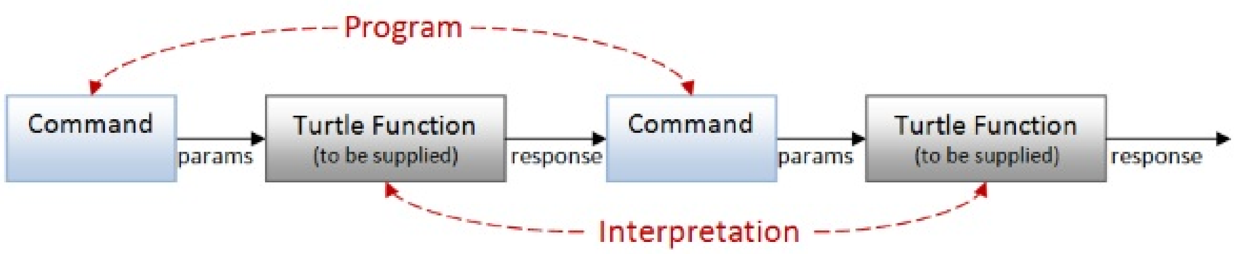
For our final approach, we'll look at a way to *completely* decouple the programming of the turtle from its interpretation.

This is similar to the batch processing using command objects approach, but is enhanced to support responding to the output of a command.

Designing an interpreter

The approach we will take is to design an "interpreter" for a set of turtle commands, where the client provides the commands to the turtle, and responds to outputs from the turtle, but the actual turtle functions are provided later by a particular implementation.

In other words, we have a chain of interleaved commands and turtle functions that look like this:



So how can we model this design in code?

For a first attempt, let's model the chain as a sequence of request/response pairs. We send a command to the turtle and it responds appropriately with a `MoveResponse` or whatever, like this:

```
// we send this to the turtle...
type TurtleCommand =
    | Move of Distance
    | Turn of Angle
    | PenUp
    | PenDown
    | SetColor of PenColor

// ... and the turtle replies with one of these
type TurtleResponse =
    | Moved of MoveResponse
    | Turned
    | PenWentUp
    | PenWentDown
    | ColorSet of SetColorResponse
```

The problem is that we cannot be sure that the response correctly matches the command. For example, if I send a `Move` command, I expect to get a `MoveResponse`, and never a `SetColorResponse`. But this implementation doesn't enforce that!

We want to make illegal states unrepresentable -- how can we do that?

The trick is to combine the request and response in *pairs*. That is, for a `Move` command, there is an associated function which is given a `MoveResponse` as input, and similarly for each other combination. Commands that have no response can be considered as returning `unit` for now.

```
Move command => pair of (Move command parameters), (function MoveResponse -> something
)
Turn command => pair of (Turn command parameters), (function unit -> something)
etc
```

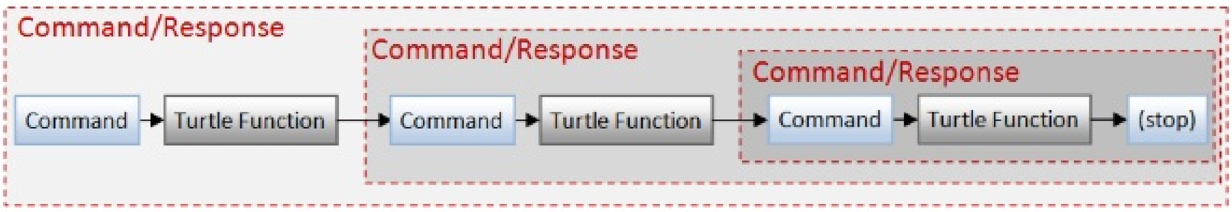
The way this works is that:

- The client creates a command, say `Move 100`, and also provides the additional function that handles the response.
- The turtle implementation for the Move command (inside the interpreter) processes the input (a `Distance`) and then generates a `MoveResponse`.
- The interpreter then takes this `MoveResponse` and calls the associated function in the pair, as supplied by the client.

By associating the `Move` command with a function in this way, we can *guarantee* that the internal turtle implementation must accept a `distance` and return a `MoveResponse`, just as we want.

The next question is: what is the `something` that is the output? It is the output after the client has handled the response -- that is, another command/response chain!

So we can model the whole chain of pairs as a recursive structure:



Or in code:

```
type TurtleProgram =
    //         (input params)  (response)
    | Move     of Distance   * (MoveResponse -> TurtleProgram)
    | Turn     of Angle      * (unit -> TurtleProgram)
    | PenUp    of (* none *)   (unit -> TurtleProgram)
    | PenDown  of (* none *)   (unit -> TurtleProgram)
    | SetColor of PenColor   * (SetColorResponse -> TurtleProgram)
```

I've renamed the type from `TurtleCommand` to `TurtleProgram` because it is no longer just a command, but is now a complete chain of commands and associated response handlers.

There's a problem though! Every step needs yet another `TurtleProgram` to follow -- so when will it stop? We need some way of saying that there is no next command.

To solve this issue, we will add a special `Stop` case to the program type:

```
type TurtleProgram =
    //         (input params)  (response)
    | Stop
    | Move     of Distance   * (MoveResponse -> TurtleProgram)
    | Turn     of Angle      * (unit -> TurtleProgram)
    | PenUp    of (* none *)   (unit -> TurtleProgram)
    | PenDown  of (* none *)   (unit -> TurtleProgram)
    | SetColor of PenColor   * (SetColorResponse -> TurtleProgram)
```

Note that there is no mention of `TurtleState` in this structure. How the turtle state is managed is internal to the interpreter, and is not part of the "instruction set", as it were.

`TurtleProgram` is an example of an Abstract Syntax Tree (AST) -- a structure that represents a program to interpreted (or compiled).

Testing the interpreter

Let's create a little program using this model. Here's our old friend `drawTriangle`:

```
let drawTriangle =
    Move (100.0, fun response ->
    Turn (120.0<Degrees>, fun () ->
    Move (100.0, fun response ->
    Turn (120.0<Degrees>, fun () ->
    Move (100.0, fun response ->
    Turn (120.0<Degrees>, fun () ->
    Stop))))))
```

This program is a data structure containing only client commands and responses -- there are no actual turtle functions in it anywhere! And yes, it is really ugly right now, but we will fix that shortly.

Now the next step is to interpret this data structure.

Let's create an interpreter that calls the real turtle functions. How would we implement the `Move` case, say?

Well, just as described above:

- Get the distance and associated function from the `Move` case
- Call the real turtle function with the distance and current turtle state, to get a `MoveResult` and a new turtle state.
- Get the next step in the program by passing the `MoveResult` to the associated function
- Finally call the interpreter again (recursively) with the new program and new turtle state.

```
let rec interpretAsTurtle state program =
    ...
    match program  with
    | Move (dist,next) ->
        let result,newState = Turtle.move log dist state
        let nextProgram = next result  // compute the next step
        interpretAsTurtle newState nextProgram
    ...
```

You can see that the updated turtle state is passed as a parameter to the next recursive call, and so no mutable field is needed.

Here's the full code for `interpretAsTurtle` :

```
let rec interpretAsTurtle state program =
    let log = printfn "%s"

    match program  with
    | Stop ->
        state
    | Move (dist,next) ->
        let result,newState = Turtle.move log dist state
        let nextProgram = next result  // compute the next step
        interpretAsTurtle newState nextProgram
    | Turn (angle,next) ->
        let newState = Turtle.turn log angle state
        let nextProgram = next()       // compute the next step
        interpretAsTurtle newState nextProgram
    | PenUp next ->
        let newState = Turtle.penUp log state
        let nextProgram = next()
        interpretAsTurtle newState nextProgram
    | PenDown next ->
        let newState = Turtle.penDown log state
        let nextProgram = next()
        interpretAsTurtle newState nextProgram
    | SetColor (color,next) ->
        let result,newState = Turtle.setColor log color state
        let nextProgram = next result
        interpretAsTurtle newState nextProgram
```

Let's run it:

```
let program = drawTriangle
let interpret = interpretAsTurtle   // choose an interpreter
let initialState = Turtle.initialTurtleState
interpret initialState program |> ignore
```

and the output is exactly what we have seen before:

```
Move 100.0
...Draw line from (0.0,0.0) to (100.0,0.0) using Black
Turn 120.0
Move 100.0
...Draw line from (100.0,0.0) to (50.0,86.6) using Black
Turn 120.0
Move 100.0
...Draw line from (50.0,86.6) to (0.0,0.0) using Black
Turn 120.0
```

But unlike all the previous approaches we can take *exactly the same program* and interpret it in a new way. We don't need to set up any kind of dependency injection, we just need to use a different interpreter.

So let's create another interpreter that aggregates the distance travelled, without caring about the turtle state:

```
let rec interpretAsDistance distanceSoFar program =
    let recurse = interpretAsDistance
    let log = printfn "%s"

    match program with
    | Stop ->
        distanceSoFar
    | Move (dist,next) ->
        let newDistanceSoFar = distanceSoFar + dist
        let result = Turtle.MoveOk   // hard-code result
        let nextProgram = next result
        recurse newDistanceSoFar nextProgram
    | Turn (angle,next) ->
        // no change in distanceSoFar
        let nextProgram = next()
        recurse distanceSoFar nextProgram
    | PenUp next ->
        // no change in distanceSoFar
        let nextProgram = next()
        recurse distanceSoFar nextProgram
    | PenDown next ->
        // no change in distanceSoFar
        let nextProgram = next()
        recurse distanceSoFar nextProgram
    | SetColor (color,next) ->
        // no change in distanceSoFar
        let result = Turtle.ColorOk   // hard-code result
        let nextProgram = next result
        recurse distanceSoFar nextProgram
```

In this case, I've aliased `interpretAsDistance` as `recurse` locally to make it obvious what kind of recursion is happening.

Let's run the same program with this new interpreter:

```
let program = drawTriangle           // same program
let interpret = interpretAsDistance  // choose an interpreter
let initialState = 0.0
interpret initialState program |> printfn "Total distance moved is %0.1f"
```

and the output is again exactly what we expect:

```
Total distance moved is 300.0
```

Creating a "turtle program" workflow

That code for creating a program to interpret was pretty ugly! Can we create a computation expression to make it look nicer?

Well, in order to create a computation expression, we need `return` and `bind` functions, and those require that the `TurtleProgram` type be generic.

No problem! Let's make `TurtleProgram` generic then:

```
type TurtleProgram<'a> =
    | Stop     of 'a
    | Move     of Distance * (MoveResponse -> TurtleProgram<'a>)
    | Turn     of Angle    * (unit -> TurtleProgram<'a>)
    | PenUp    of            (unit -> TurtleProgram<'a>)
    | PenDown  of            (unit -> TurtleProgram<'a>)
    | SetColor of PenColor * (SetColorResponse -> TurtleProgram<'a>)
```

Note that the `Stop` case has a value of type `'a` associated with it now. This is needed so that we can implement `return` properly:

```
let returnT x =
    Stop x
```

The `bind` function is more complicated to implement. Don't worry about how it works right now -- the important thing is that the types match up and it compiles!

```
let rec bindT f inst  =
    match inst with
    | Stop x ->
        f x
    | Move(dist,next) ->
        (*
        Move(dist,fun moveResponse -> (bindT f)(next moveResponse))
        *)
        // "next >> bindT f" is a shorter version of function response
        Move(dist,next >> bindT f)
    | Turn(angle,next) ->
        Turn(angle,next >> bindT f)
    | PenUp(next) ->
        PenUp(next >> bindT f)
    | PenDown(next) ->
        PenDown(next >> bindT f)
    | SetColor(color,next) ->
        SetColor(color,next >> bindT f)
```

With `bind` and `return` in place, we can create a computation expression:

```
// define a computation expression builder
type TurtleProgramBuilder() =
    member this.Return(x) = returnT x
    member this.Bind(x,f) = bindT f x
    member this.Zero(x) = returnT ()

// create an instance of the computation expression builder
let turtleProgram = TurtleProgramBuilder()
```

We can now create a workflow that handles `MoveResponse` s just as in the monadic control flow example (way 12) earlier.

```
// helper functions
let stop = fun x -> Stop x
let move dist  = Move (dist, stop)
let turn angle  = Turn (angle, stop)
let penUp  = PenUp stop
let penDown  = PenDown stop
let setColor color = SetColor (color,stop)

let handleMoveResponse log moveResponse = turtleProgram {
    match moveResponse with
    | Turtle.MoveOk ->
        ()
    | Turtle.HitABarrier ->
        // turn 90 before trying again
        log "Oops -- hit a barrier -- turning"
        let! x = turn 90.0<Degrees>
        ()
    }

// example
let drawTwoLines log = turtleProgram {
    let! response = move 60.0
    do! handleMoveResponse log response
    let! response = move 60.0
    do! handleMoveResponse log response
    }
```

Let's interpret this using the real turtle functions (assuming that the `interpretAsTurtle` function has been modified to handle the new generic structure):

```
let log = printfn "%s"
let program = drawTwoLines log
let interpret = interpretAsTurtle
let initialState = Turtle.initialTurtleState
interpret initialState program |> ignore
```

The output shows that the `MoveResponse` is indeed being handled correctly when the barrier is encountered:

```
Move 60.0
...Draw line from (0.0,0.0) to (60.0,0.0) using Black
Move 60.0
...Draw line from (60.0,0.0) to (100.0,0.0) using Black
Oops -- hit a barrier -- turning
Turn 90.0
```

Refactoring the `TurtleProgram` type into two parts

This approach works fine, but it bothers me that there is a special `Stop` case in the `TurtleProgram` type. It would nice if we could somehow just focus on the five turtle actions and ignore it.

As it turns out, there *is* a way to do this. In Haskell and Scalaz it would be called a "free monad", but since F# doesn't support typeclasses, I'll just call it the "free monad pattern" that you can use to solve the problem. There's a little bit of boilerplate that you have to write, but not much.

The trick is to separate the api cases and "stop"/"keep going" logic into two separate types, like this:

```
/// Create a type to represent each instruction
type TurtleInstruction<'next> =
    | Move     of Distance * (MoveResponse -> 'next)
    | Turn     of Angle    * 'next
    | PenUp    of            'next
    | PenDown  of            'next
    | SetColor of PenColor * (SetColorResponse -> 'next)

/// Create a type to represent the Turtle Program
type TurtleProgram<'a> =
    | Stop of 'a
    | KeepGoing of TurtleInstruction<TurtleProgram<'a>>
```

Note that I've also changed the responses for `Turn` , `PenUp` and `PenDown` to be single values rather than a unit function. `Move` and `SetColor` remain as functions though.

In this new "free monad" approach, the only custom code we need to write is a simple `map` function for the api type, in this case `TurtleInstruction` :

```
let mapInstr f inst  =
    match inst with
    | Move(dist,next) ->      Move(dist,next >> f)
    | Turn(angle,next) ->     Turn(angle,f next)
    | PenUp(next) ->          PenUp(f next)
    | PenDown(next) ->        PenDown(f next)
    | SetColor(color,next) -> SetColor(color,next >> f)
```

The rest of the code (`return` , `bind` , and the computation expression) is always implemented exactly the same way, regardless of the particular api. That is, more boilerplate is needed but less thinking is required!

The interpreters need to change in order to handle the new cases. Here's a snippet of the new version of `interpretAsTurtle` :

```
let rec interpretAsTurtle log state program =
    let recurse = interpretAsTurtle log

    match program with
    | Stop a ->
        state
    | KeepGoing (Move (dist,next)) ->
        let result,newState = Turtle.move log dist state
        let nextProgram = next result // compute next program
        recurse newState nextProgram
    | KeepGoing (Turn (angle,next)) ->
        let newState = Turtle.turn log angle state
        let nextProgram = next        // use next program directly
        recurse newState nextProgram
```

And we also need to adjust the helper functions when creating a workflow. You can see below that we now have slightly more complicated code like `KeepGoing (Move (dist, Stop))` instead of the simpler code in the original interpreter.

```
// helper functions
let stop = Stop()
let move dist  = KeepGoing (Move (dist, Stop))    // "Stop" is a function
let turn angle  = KeepGoing (Turn (angle, stop))  // "stop" is a value
let penUp  = KeepGoing (PenUp stop)
let penDown  = KeepGoing (PenDown stop)
let setColor color = KeepGoing (SetColor (color,Stop))

let handleMoveResponse log moveResponse = turtleProgram {
    ... // as before

// example
let drawTwoLines log = turtleProgram {
    let! response = move 60.0
    do! handleMoveResponse log response
    let! response = move 60.0
    do! handleMoveResponse log response
    }
```

But with those changes, we are done, and the code works just as before.

Advantages and disadvantages of the interpreter pattern

Advantages

- *Decoupling.* An abstract syntax tree completely decouples the program flow from the implementation and allows lots of flexibility.
- *Optimization*. Abstract syntax trees can be manipulated and changed *before* running

them, in order to do optimizations or other transformations. As an example, for the turtle program, we could process the tree and collapse all contiguous sequences of `Turn` into a single `Turn` operation. This is a simple optimization which saves on the number of times we need to communicate with a physical turtle. Twitter's Stitch library does something like this, but obviously, in a more sophisticated way. This video has a good explanation.

- *Minimal code for a lot of power*. The "free monad" approach to creating abstract syntax trees allows you to focus on the API and ignore the Stop/KeepGoing logic, and also means that only a minimal amount of code needs to be customized. For more on the free monad, start with this excellent video and then see this post and this one.

Disadvantages

- Complex to understand.
- Only works well if there are a limited set of operations to perform.
- Can be inefficient if the ASTs get too large.

The source code for this version is available here (original version) and here ("free monad" version).

Review of techniques used

In this post, we looked at thirteen different ways to implement a turtle API, using a wide variety of different techniques. Let's quickly run down all the techniques that were used:

- **Pure, stateless functions**. As seen in all of the FP-oriented examples. All these are very easy to test and mock.
- **Partial application**. As first seen in the simplest FP example (way 2), when the turtle functions had the logging function applied so that the main flow could use piping, and thereafter used extensively, particularly in the "dependency injection using functions approach" (way 7).
- **Object expressions** to implement an interface without creating a class, as seen in way 6.
- **The Result type** (a.k.a the Either monad). Used in all the functional API examples (e.g. way 4) to return an error rather than throw an exception.
- **Applicative "lifting"** (e.g. `lift2`) to lift normal functions to the world of `Result`s, again in way 4 and others.
- **Lots of different ways of managing state**:
    - mutable fields (way 1)
    - managing state explicitly and piping it though a series of functions (way 2)

  - having state only at the edge (the functional core/imperative shell in way 4)
  - hiding state in an agent (way 5)
  - threading state behind the scenes in a state monad (the `turtle` workflow in ways 8 and 12)
  - avoiding state altogether by using batches of commands (way 9) or batches of events (way 10) or an interpreter (way 13)
- **Wrapping a function in a type**. Used in way 8 to manage state (the State monad) and in way 13 to store responses.
- **Computation expressions**, lots of them! We created and used three:
  - `result` for working with errors
  - `turtle` for managing turtle state
  - `turtleProgram` for building an AST in the interpreter approach (way 13).
- **Chaining of monadic functions** in the `result` and `turtle` workflows. The underlying functions are monadic ("diagonal") and would not normally compose properly, but inside a workflow, they can be sequenced easily and transparently.
- **Representing behavior as a data structure** in the "functional dependency injection" example (way 7) so that a single function could be passed in rather than a whole interface.
- **Decoupling using a data-centric protocol** as seen in the agent, batch command, event sourcing, and interpreter examples.
- **Lock free and async processing** using an agent (way 5).
- **The separation of "building" a computation vs. "running" it**, as seen in the `turtle` workflows (ways 8 and 12) and the `turtleProgram` workflow (way 13: interpreter).
- **Use of event sourcing to rebuild state** from scratch rather than maintaining mutable state in memory, as seen in the event sourcing (way 10) and FRP (way 11) examples.
- **Use of event streams** and FRP (way 11) to break business logic into small, independent, and decoupled processors rather than having a monolithic object.

I hope it's clear that examining these thirteen ways is just a fun exercise, and I'm not suggesting that you immediately convert all your code to use stream processors and interpreters! And, especially if you are working with people who are new to functional programming, I would tend to stick with the earlier (and simpler) approaches unless there is a clear benefit in exchange for the extra complexity.

Summary

> When the tortoise crawled out of sight,
> It marked the edge
> Of one of many circles.
> -- *"Thirteen ways of looking at a turtle", by Wallace D Coriacea*

I hope you enjoyed this post. I certainly enjoyed writing it. As usual, it ended up much longer than I intended, so I hope that the effort of reading it was worth it to you!

If you like this kind of comparative approach, and want more, check out the posts by Yan Cui, who is doing something similar on his blog.

Enjoy the rest of the F# Advent Calendar. Happy Holidays!

The source code for this post is available on github.

Thirteen ways of looking at a turtle - addendum

In this, the third part of my two-part mega-post, I'm continuing to stretch the simple turtle graphics model to the breaking point.

In the first and second post, I described thirteen different ways of looking at a turtle graphics implementation.

Unfortunately, after I published them, I realized that there were some other ways that I had forgotten to mention. So in this post, you'll get to see two BONUS ways.

- Way 14. Abstract Data Turtle, in which we encapsulate the details of a turtle implementation by using an Abstract Data Type.
- Way 15. Capability-based Turtle, in which we control what turtle functions are available to a client, based on the current state of the turtle.

As a reminder, here were the previous thirteen ways:

- Way 1. A basic object-oriented approach, in which we create a class with mutable state.
- Way 2. A basic functional approach, in which we create a module of functions with immutable state.
- Way 3. An API with a object-oriented core, in which we create an object-oriented API that calls a stateful core class.
- Way 4. An API with a functional core, in which we create an stateful API that uses stateless core functions.
- Way 5. An API in front of an agent, in which we create an API that uses a message queue to communicate with an agent.
- Way 6. Dependency injection using interfaces, in which we decouple the implementation from the API using an interface or record of functions.
- Way 7. Dependency injection using functions, in which we decouple the implementation from the API by passing a function parameter.
- Way 8. Batch processing using a state monad, in which we create a special "turtle workflow" computation expression to track state for us.
- Way 9. Batch processing using command objects, in which we create a type to represent a turtle command, and then process a list of commands all at once.
- Interlude: Conscious decoupling with data types. A few notes on using data vs. interfaces for decoupling.
- Way 10. Event sourcing, in which state is built from a list of past events.
- Way 11. Functional Retroactive Programming (stream processing), in which business

logic is based on reacting to earlier events.
- Episode V: The Turtle Strikes Back, in which the turtle API changes so that some commands may fail.
- Way 12. Monadic control flow, in which we make decisions in the turtle workflow based on results from earlier commands.
- Way 13. A turtle interpreter, in which we completely decouple turtle programming from turtle implementation, and nearly encounter the free monad.
- Review of all the techniques used.

All source code for this post is available on github.

14: Abstract Data Turtle

In this design, we use the concept of an abstract data type to encapsulate the operations on a turtle.

That is, a "turtle" is defined as an opaque type along with a corresponding set of operations, in the same way that standard F# types such as `List` , `Set` and `Map` are defined.

That is, we have number of functions that work on the type, but we are not allowed to see "inside" the type itself.

In a sense, you can think of it as a third alternative to the OO approach in way 1 and the functional approach in way 2.

- In the OO implementation, the details of the internals are nicely encapsulated, and access is only via methods. The downside of the OO class is that it is mutable.
- In the FP implementation, the `TurtleState` is immutable, but the downside is that the internals of the state are public, and some clients may have accessed these fields, so if we ever change the design of `TurtleState` , these clients may break.

The abstract data type implementation combines the best of both worlds: the turtle state is immutable, as in the original FP way, but no client can access it, as in the OO way.

The design for this (and for any abstract type) is as follows:

- The turtle state type itself is public, but its constructor and fields are private.
- The functions in the associated `Turtle` module can see inside the turtle state type (and so are unchanged from the FP design).
- Because the turtle state constructor is private, we need a constructor function in the `Turtle` module.
- The client can *not* see inside the turtle state type, and so must rely entirely on the

`Turtle` module functions.

That's all there is to it. We only need to add some privacy modifiers to the earlier FP version and we are done!

The implementation

First, we are going to put both the turtle state type and the `Turtle` module inside a common module called `AdtTurtle`. This allows the turtle state to be accessible to the functions in the `AdtTurtle.Turtle` module, while being inaccessible outside the `AdtTurtle`.

Next, the turtle state type is going to be called `Turtle` now, rather than `TurtleState`, because we are treating it almost as an object.

Finally, the associated module `Turtle` (that contains the functions) is going have some special attributes:

- `RequireQualifiedAccess` means the module name *must* be used when accessing the functions (just like `List` module)
- `ModuleSuffix` is needed so the that module can have the same name as the state type. This would not be required for generic types (e.g if we had `Turtle<'a>` instead).

```
module AdtTurtle =

    /// A private structure representing the turtle
    type Turtle = private {
        position : Position
        angle : float<Degrees>
        color : PenColor
        penState : PenState
    }

    /// Functions for manipulating a turtle
    /// "RequireQualifiedAccess" means the module name *must*
    ///    be used (just like List module)
    /// "ModuleSuffix" is needed so the that module can
    ///    have the same name as the state type
    [<RequireQualifiedAccess>]
    [<CompilationRepresentation (CompilationRepresentationFlags.ModuleSuffix)>]
    module Turtle =
```

An alternative way to avoid collisions is to have the state type have a different case, or a different name with a lowercase alias, like this:

```
type TurtleState = { ... }
type turtle = TurtleState

module Turtle =
    let something (t:turtle) = t
```

No matter how the naming is done, we will need a way to construct a new `Turtle`.

If there are no parameters to the constructor, and the state is immutable, then we just need an initial value rather than a function (like `Set.empty` say).

Otherwise we can define a function called `make` (or `create` or similar):

```
[<RequireQualifiedAccess>]
[<CompilationRepresentation (CompilationRepresentationFlags.ModuleSuffix)>]
module Turtle =

    /// return a new turtle with the specified color
    let make(initialColor) = {
        position = initialPosition
        angle = 0.0<Degrees>
        color = initialColor
        penState = initialPenState
    }
```

The rest of the turtle module functions are unchanged from their implementation in way 2.

An ADT client

Let's look the client now.

First, let's check that the state really is private. If we try to create a state explicitly, as shown below, we get a compiler error:

```
let initialTurtle = {
    position = initialPosition
    angle = 0.0<Degrees>
    color = initialColor
    penState = initialPenState
}
// Compiler error FS1093:
//    The union cases or fields of the type 'Turtle'
//    are not accessible from this code location
```

If we use the constructor and then try to directly access a field directly (such as `position`), we again get a compiler error:

```
let turtle = Turtle.make(Red)
printfn "%A" turtle.position
// Compiler error FS1093:
//    The union cases or fields of the type 'Turtle'
//    are not accessible from this code location
```

But if we stick to the functions in the `Turtle` module, we can safely create a state value and then call functions on it, just as we did before:

```
// versions with log baked in (via partial application)
let move = Turtle.move log
let turn = Turtle.turn log
// etc

let drawTriangle() =
    Turtle.make(Red)
    |> move 100.0
    |> turn 120.0<Degrees>
    |> move 100.0
    |> turn 120.0<Degrees>
    |> move 100.0
    |> turn 120.0<Degrees>
```

Advantages and disadvantages of ADTs

Advantages

- All code is stateless, hence easy to test.
- The encapsulation of the state means that the focus is always fully on the behavior and properties of the type.
- Clients can never have a dependency on a particular implementation, which means that implementations can be changed safely.
- You can even swap implementations (e.g. by shadowing, or linking to a different assembly) for testing, performance, etc.

Disadvantages

- The client has to manage the current turtle state.
- The client has no control over the implementation (e.g. by using dependency injection).

For more on ADTs in F#, see this talk and thread by Bryan Edds.

The source code for this version is available here.

15: Capability-based Turtle

In the "monadic control flow" approach (way 12) we handled responses from the turtle telling us that it had hit a barrier.

But even though we had hit a barrier, nothing was stopping us from calling the `move` operation over and over again!

Now imagine that, once we had hit the barrier, the `move` operation was no longer available to us. We couldn't abuse it because it would be no longer there!

To make this work, we shouldn't provide an API, but instead, after each call, return a list of functions that the client can call to do the next step. The functions would normally include the usual suspects of `move`, `turn`, `penUp`, etc., but when we hit a barrier, `move` would be dropped from that list. Simple, but effective.

This technique is closely related to an authorization and security technique called *capability-based security*. If you are interested in learning more, I have a whole series of posts devoted to it.

Designing a capability-based Turtle

The first thing is to define the record of functions that will be returned after each call:

```
type MoveResponse =
    | MoveOk
    | HitABarrier

type SetColorResponse =
    | ColorOk
    | OutOfInk

type TurtleFunctions = {
    move     : MoveFn option
    turn     : TurnFn
    penUp    : PenUpDownFn
    penDown  : PenUpDownFn
    setBlack : SetColorFn  option
    setBlue  : SetColorFn  option
    setRed   : SetColorFn  option
    }
and MoveFn =      Distance -> (MoveResponse * TurtleFunctions)
and TurnFn =      Angle    -> TurtleFunctions
and PenUpDownFn = unit     -> TurtleFunctions
and SetColorFn =  unit     -> (SetColorResponse * TurtleFunctions)
```

Let's look at these declarations in detail.

First, there is no `TurtleState` anywhere. The published turtle functions will encapsulate the state for us. Similarly there is no `log` function.

Next, the record of functions `TurtleFunctions` defines a field for each function in the API (`move` , `turn` , etc.):

- The `move` function is optional, meaning that it might not be available.
- The `turn` , `penUp` and `penDown` functions are always available.
- The `setColor` operation has been broken out into three separate functions, one for each color, because you might not be able to use red ink, but still be able to use blue ink. To indicate that these functions might not be available, `option` is used again.

We have also declared type aliases for each function to make them easier to work. Writing `MoveFn` is easier than writing `Distance -> (MoveResponse * TurtleFunctions)` everywhere! Note that, since these definitions are mutually recursive, I was forced to use the `and` keyword.

Finally, note the difference between the signature of `MoveFn` in this design and the signature of `move` in the earlier design of way 12.

Earlier version:

```
val move :
    Log -> Distance -> TurtleState -> (MoveResponse * TurtleState)
```

New version:

```
val move :
    Distance -> (MoveResponse * TurtleFunctions)
```

On the input side, the `Log` and `TurtleState` parameters are gone, and on the output side, the `TurtleState` has been replaced with `TurtleFunctions` .

This means that somehow, the output of every API function must be changed to be a `TurtleFunctions` record.

Implementing the turtle operations

In order to decide whether we can indeed move, or use a particular color, we first need to augment the `TurtleState` type to track these factors:

```
type Log = string -> unit

type private TurtleState = {
    position : Position
    angle : float<Degrees>
    color : PenColor
    penState : PenState

    canMove : bool                // new!
    availableInk: Set<PenColor>   // new!
    logger : Log                  // new!
}
```

This has been enhanced with

- `canMove` , which if false means that we are at a barrier and should not return a valid `move` function.
- `availableInk` contains a set of colors. If a color is not in this set, then we should not return a valid `setColorXXX` function for that color.
- Finally, we've added the `log` function into the state so that we don't have to pass it explicitly to each operation. It will get set once, when the turtle is created.

The `TurtleState` is getting a bit ugly now, but that's alright, because it's private! The clients will never even see it.

With this augmented state available, we can change `move` . First we'll make it private, and second we'll set the `canMove` flag (using `moveResult <> HitABarrier`) before returning a new state:

```
/// Function is private! Only accessible to the client via the TurtleFunctions record
let private move log distance state =

    log (sprintf "Move %0.1f" distance)
    // calculate new position
    let newPosition = calcNewPosition distance state.angle state.position
    // adjust the new position if out of bounds
    let moveResult, newPosition = checkPosition newPosition
    // draw line if needed
    if state.penState = Down then
        dummyDrawLine log state.position newPosition state.color

    // return the new state and the Move result
    let newState = {
        state with
         position = newPosition
         canMove = (moveResult <> HitABarrier)   // NEW!
        }
    (moveResult,newState)
```

We need some way of changing `canMove` back to true! So let's assume that if you turn, you can move again.

Let's add that logic to the `turn` function then:

```
let private turn log angle state =
    log (sprintf "Turn %0.1f" angle)
    // calculate new angle
    let newAngle = (state.angle + angle) % 360.0<Degrees>
    // NEW!! assume you can always move after turning
    let canMove = true
    // update the state
    {state with angle = newAngle; canMove = canMove}
```

The `penUp` and `penDown` functions are unchanged, other than being made private.

And for the last operation, `setColor`, we'll remove the ink from the availability set as soon as it is used just once!

```
let private setColor log color state =
    let colorResult =
        if color = Red then OutOfInk else ColorOk
    log (sprintf "SetColor %A" color)

    // NEW! remove color ink from available inks
    let newAvailableInk = state.availableInk |> Set.remove color

    // return the new state and the SetColor result
    let newState = {state with color = color; availableInk = newAvailableInk}
    (colorResult,newState)
```

Finally we need a function that can create a `TurtleFunctions` record from the `TurtleState`. I'll call it `createTurtleFunctions`.

Here's the complete code, and I'll discuss it in detail below:

```
/// Create the TurtleFunctions structure associated with a TurtleState
let rec private createTurtleFunctions state =
    let ctf = createTurtleFunctions  // alias

    // create the move function,
    // if the turtle can't move, return None
    let move =
        // the inner function
        let f dist =
            let resp, newState = move state.logger dist state
            (resp, ctf newState)
```

```
        // return Some of the inner function
        // if the turtle can move, or None
        if state.canMove then
            Some f
        else
            None

    // create the turn function
    let turn angle =
        let newState = turn state.logger angle state
        ctf newState

    // create the pen state functions
    let penDown() =
        let newState = penDown state.logger state
        ctf newState

    let penUp() =
        let newState = penUp state.logger state
        ctf newState

    // create the set color functions
    let setColor color =
        // the inner function
        let f() =
            let resp, newState = setColor state.logger color state
            (resp, ctf newState)

        // return Some of the inner function
        // if that color is available, or None
        if state.availableInk |> Set.contains color then
            Some f
        else
            None

    let setBlack = setColor Black
    let setBlue = setColor Blue
    let setRed = setColor Red

    // return the structure
    {
    move     = move
    turn     = turn
    penUp    = penUp
    penDown  = penDown
    setBlack = setBlack
    setBlue  = setBlue
    setRed   = setRed
    }
```

Let's look at how this works.

First, note that this function needs the `rec` keyword attached, as it refers to itself. I've added a shorter alias (`ctf`) for it as well.

Next, new versions of each of the API functions are created. For example, a new `turn` function is defined like this:

```
let turn angle =
    let newState = turn state.logger angle state
    ctf newState
```

This calls the original `turn` function with the logger and state, and then uses the recursive call (`ctf`) to convert the new state into the record of functions.

For an optional function like `move` , it is a bit more complicated. An inner function `f` is defined, using the orginal `move` , and then either `f` is returned as `Some` , or `None` is returned, depending on whether the `state.canMove` flag is set:

```
// create the move function,
// if the turtle can't move, return None
let move =
    // the inner function
    let f dist =
        let resp, newState = move state.logger dist state
        (resp, ctf newState)

    // return Some of the inner function
    // if the turtle can move, or None
    if state.canMove then
        Some f
    else
        None
```

Similarly, for `setColor` , an inner function `f` is defined and then returned or not depending on whether the color parameter is in the `state.availableInk` collection:

```
let setColor color =
    // the inner function
    let f() =
        let resp, newState = setColor state.logger color state
        (resp, ctf newState)

    // return Some of the inner function
    // if that color is available, or None
    if state.availableInk |> Set.contains color then
        Some f
    else
        None
```

Finally, all these functions are added to the record:

```
// return the structure
{
move     = move
turn     = turn
penUp    = penUp
penDown  = penDown
setBlack = setBlack
setBlue  = setBlue
setRed   = setRed
}
```

And that's how you build a `TurtleFunctions` record!

We need one more thing: a constructor to create some initial value of the `TurtleFunctions`, since we no longer have direct access to the API. This is now the ONLY public function available to the client!

```
/// Return the initial turtle.
/// This is the ONLY public function!
let make(initialColor, log) =
    let state = {
        position = initialPosition
        angle = 0.0<Degrees>
        color = initialColor
        penState = initialPenState
        canMove = true
        availableInk = [Black; Blue; Red] |> Set.ofList
        logger = log
    }
    createTurtleFunctions state
```

This function bakes in the `log` function, creates a new state, and then calls `createTurtleFunctions` to return a `TurtleFunction` record for the client to use.

Implementing a client of the capability-based turtle

Let's try using this now. First, let's try to do `move 60` and then `move 60` again. The second move should take us to the boundary (at 100), and so at that point the `move` function should no longer be available.

First, we create the `TurtleFunctions` record with `Turtle.make`. Then we can't just move immediately, we have to test to see if the `move` function is available first:

```
let testBoundary() =
    let turtleFns = Turtle.make(Red,log)
    match turtleFns.move with
    | None ->
        log "Error: Can't do move 1"
    | Some moveFn ->
        ...
```

In the last case, the `moveFn` is available, so we can call it with a distance of 60.

The output of the function is a pair: a `MoveResponse` type and a new `TurtleFunctions` record.

We'll ignore the `MoveResponse` and check the `TurtleFunctions` record again to see if we can do the next move:

```
let testBoundary() =
    let turtleFns = Turtle.make(Red,log)
    match turtleFns.move with
    | None ->
        log "Error: Can't do move 1"
    | Some moveFn ->
        let (moveResp,turtleFns) = moveFn 60.0
        match turtleFns.move with
        | None ->
            log "Error: Can't do move 2"
        | Some moveFn ->
            ...
```

And finally, one more time:

```
let testBoundary() =
    let turtleFns = Turtle.make(Red,log)
    match turtleFns.move with
    | None ->
        log "Error: Can't do move 1"
    | Some moveFn ->
        let (moveResp,turtleFns) = moveFn 60.0
        match turtleFns.move with
        | None ->
            log "Error: Can't do move 2"
        | Some moveFn ->
            let (moveResp,turtleFns) = moveFn 60.0
            match turtleFns.move with
            | None ->
                log "Error: Can't do move 3"
            | Some moveFn ->
                log "Success"
```

If we run this, we get the output:

```
Move 60.0
...Draw line from (0.0,0.0) to (60.0,0.0) using Red
Move 60.0
...Draw line from (60.0,0.0) to (100.0,0.0) using Red
Error: Can't do move 3
```

Which shows that indeed, the concept is working!

That nested option matching is really ugly, so let's whip up a quick `maybe` workflow to make it look nicer:

```
type MaybeBuilder() =
    member this.Return(x) = Some x
    member this.Bind(x,f) = Option.bind f x
    member this.Zero() = Some()
let maybe = MaybeBuilder()
```

And a logging function that we can use inside the workflow:

```
/// A function that logs and returns Some(),
/// for use in the "maybe" workflow
let logO message =
    printfn "%s" message
    Some ()
```

Now we can try setting some colors using the `maybe` workflow:

```
let testInk() =
    maybe {
    // create a turtle
    let turtleFns = Turtle.make(Black,log)

    // attempt to get the "setRed" function
    let! setRedFn = turtleFns.setRed

    // if so, use it
    let (resp,turtleFns) = setRedFn()

    // attempt to get the "move" function
    let! moveFn = turtleFns.move

    // if so, move a distance of 60 with the red ink
    let (resp,turtleFns) = moveFn 60.0

    // check if the "setRed" function is still available
    do! match turtleFns.setRed with
        | None ->
            logO "Error: Can no longer use Red ink"
        | Some _ ->
            logO "Success: Can still use Red ink"

    // check if the "setBlue" function is still available
    do! match turtleFns.setBlue with
        | None ->
            logO "Error: Can no longer use Blue ink"
        | Some _ ->
            logO "Success: Can still use Blue ink"

    } |> ignore
```

The output of this is:

```
SetColor Red
Move 60.0
...Draw line from (0.0,0.0) to (60.0,0.0) using Red
Error: Can no longer use Red ink
Success: Can still use Blue ink
```

Actually, using a `maybe` workflow is not a very good idea, because the first failure exits the workflow! You'd want to come up with something a bit better for real code, but I hope that you get the idea.

Advantages and disadvantages of a capability-based approach

Advantages

- Prevents clients from abusing the API.
- Allows APIs to evolve (and devolve) without affecting clients. For example, I could transition to a monochrome-only turtle by hard-coding `None` for each color function in the record of functions, after which I could safely remove the `setColor` implementation. During this process no client would break! This is similar to the HATEAOS approach for RESTful web services.
- Clients are decoupled from a particular implementation because the record of functions acts as an interface.

Disadvantages

- Complex to implement.
- The client's logic is much more convoluted as it can never be sure that a function will be available! It has to check every time.
- The API is not easily serializable, unlike some of the data-oriented APIs.

For more on capability-based security, see my posts or watch my "Enterprise Tic-Tac-Toe" video.

The source code for this version is available here.

Summary

> I was of three minds,
> Like a finger tree
> In which there are three immutable turtles.
> -- *"Thirteen ways of looking at a turtle", by Wallace D Coriacea*

I feel better now that I've got these two extra ways out of my system! Thanks for reading!

The source code for this post is available on github.

.

How to design and code a complete program

"I think I understand functional programming at the micro level, and I have written toy programs, but how do I actually go about writing a complete application, with real data, real error handling, and so on?"

This is a very common question, so I thought that in this series of posts I'd describe a recipe for doing exactly this, covering design, validation, error handling, persistence, dependency management, code organization, and so on.

Some comments and caveats first:

- I'll focus on just a single use case rather than a whole application. I hope that it will be obvious how to extend the code as needed.
- This will deliberately be a very simple *data-flow oriented* recipe with no special tricks or advanced techniques. But if you are just getting started, I think it is useful to have some straightforward steps you can follow to get a predictable result. I don't claim that this is the one true way of doing this. Different scenarios will need different recipes, and of course, as you get more expert, you may find this recipe too simplistic and limited.
- To help ease the transition from object-oriented design, I will try to use familiar concepts such as "patterns", "services", "dependency injection", and so on, and explain how they map to functional concepts.
- This recipe is also deliberately somewhat imperative, that is, it uses an explicit step-by-step workflow. I hope this approach will ease the transition from OO to FP.
- To keep things simple (and usable from a simple F# script), I'll mock the entire infrastructure and avoid the UI directly.

Overview

Here's an overview of what I plan to cover in this series:

- **Converting a use-case into a function**. In this first post, we'll examine a simple use case and see how it might be implemented using a functional approach.
- **Connecting smaller functions together**. In the next post, we'll discuss a simple metaphor for combining smaller functions into bigger functions.
- **Type driven design and failure types**. In the third post, we'll build the types needed for the use case, and discuss the use of special error types for the failure path.
- **Configuration and dependency management**. In this post, we'll talk about how to

wire up all the functions.

- **Validation**. In this post, we'll discuss various ways of implementing validation, and converting from the unsafe outside world to the warm fuzzy world of type safety.
- **Infrastructure**. In this post, we'll discuss various infrastructure components, such as logging, working with external code, and so on.
- **The domain layer**. In this post, we'll discuss how domain driven design works in a functional environment.
- **The presentation layer**. In this post, we'll discuss how to convey results and errors back to the UI.
- **Dealing with changing requirements**. In this post, we'll discuss how to deal with changing requirements and the effect this has on the code.

Getting started

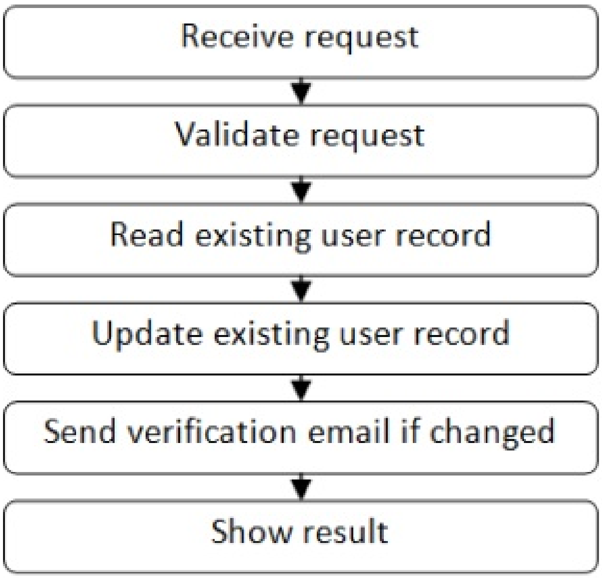
Let's pick a very simple use case, namely updating some customer information via a web service.

So here are the basic requirements:

- A user submits some data (userid, name and email).
- We check to see that the name and email are valid.
- The appropriate user record in a database is updated with the new name and email.
- If the email has changed, send a verification email to that address.
- Display the result of the operation to the user.

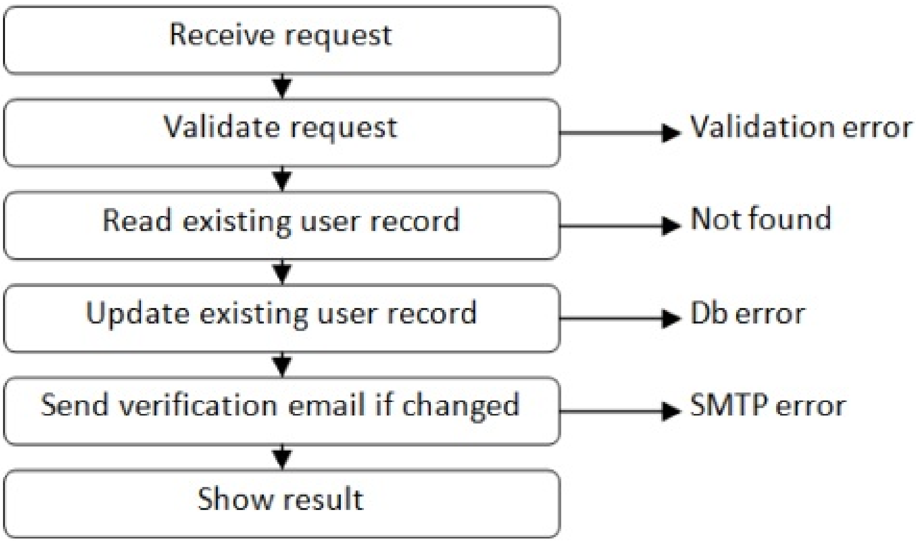
This is a typical *data centric* use case. There is some sort of request that triggers the use case, and then the request data "flows" through the system, being processed by each step in turn. This kind of scenario is common in enterprise software, which is why I am using it as an example.

Here's a diagram of the various components:



But this describes the "happy path" only. Reality is never so simple! What happens if the userid is not found in the database, or the email address is not valid, or the database has an error?

Let's update the diagram to show all the things that could go wrong.



At each step in the use case, various things could cause errors, as shown. Explaining how to handle these errors in an elegant way will be one of the goals of this series.

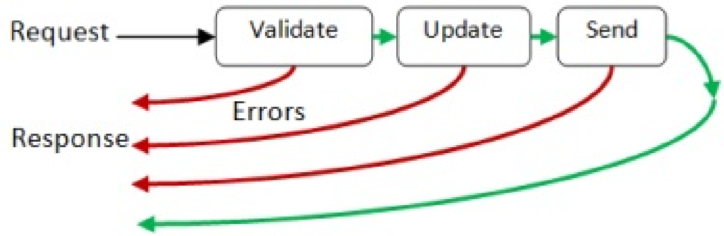
Thinking functionally

So now that we understand the steps in the use case, how do we design a solution using a functional approach?

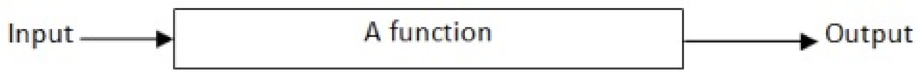
First of all, we have to address a mismatch between the original use case and functional thinking.

In the use case, we typically think of a request/response model. The request is sent, and the response comes back. If something goes wrong, the flow is short-circuited and response is returned "early".

Here's a diagram showing what I mean, based on a simplified version of the use case:



But in the functional model, a function is a black box with an input and an output, like this:

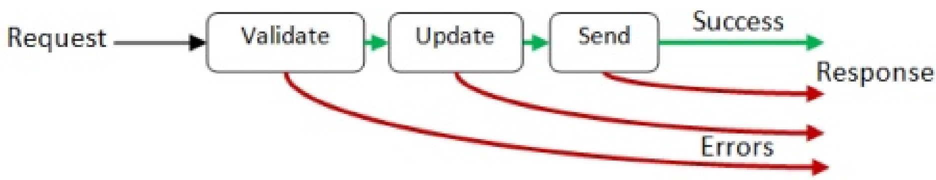


How can we adapt the use case to fit this model?

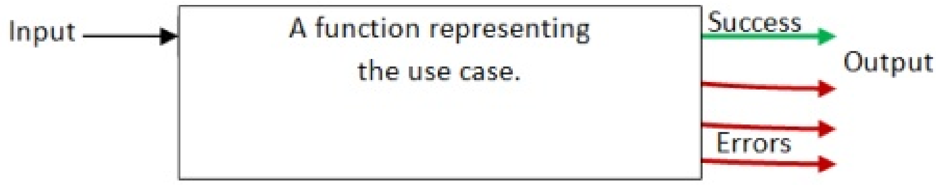
Forward flow only

First, you must recognize that functional data flow is *forward only*. You cannot do a U-turn or return early.

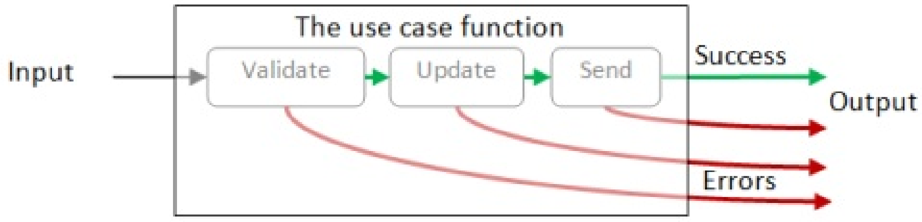
In our case, that means that all the errors *must* be sent forward to the end, as an alternative path to the happy path.



Once we have done that, we can convert the whole flow into a single "black box" function like this:



But of course, if you look inside the big function, it is made up of ("composed from" in functional-speak) smaller functions, one for each step, joined in a pipeline.



Error handling

In that last diagram, there is one success output and three error outputs. This is a problem, as functions can have only *one* output, not four!

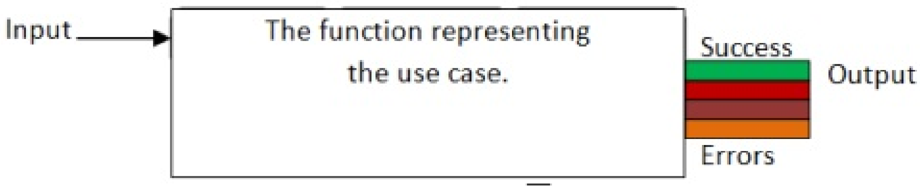
How can we handle this?

The answer is to use a union type, with one case to represent each of the different possible outputs. And then the function as a whole would indeed only have a single output.

Here's an example of a possible type definition for the output:

```
type UseCaseResult =
    | Success
    | ValidationError
    | UpdateError
    | SmtpError
```

And here's the diagram reworked to show a single output with four different cases embedded in it:

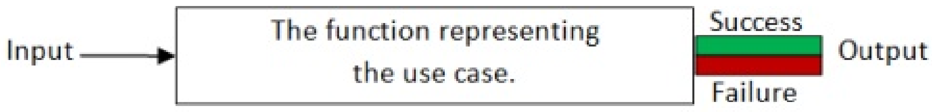


Simplifying the error handling

This does solve the problem, but having one error case for each step in the flow is brittle and not very reusable. Can we do better?

Yes! All we *really* need is *two* cases. One for the happy path, and one for all other error paths, like this:

```
type UseCaseResult =
    | Success
    | Failure
```



This type is very generic and will work with *any* workflow! In fact, you'll soon see that we can create a nice library of useful functions that will work with this type, and which can be reused for all sorts of scenarios.

One more thing though -- as it stands there is no data in the result at all, just a success/failure status. We need to tweak it a bit so that it can contain an actual success or failure object. We will specify the success type and failure type using generics (a.k.a. type parameters).

Here's the final, completely generic and reusable version:

```
type Result<'TSuccess,'TFailure> =
    | Success of 'TSuccess
    | Failure of 'TFailure
```

In fact, there is already a type almost exactly like this defined in the F# library. It's called Choice. For clarity though, I will continue to use the `Result` type defined above for this and the next post. When we come to some more serious coding, we'll revisit this.

So, now, showing the individual steps again, we can see that we will have to combine the errors from each step onto to a single "failure" path.



How to do this will be the topic of the next post.

Summary and guidelines

So far then, we have the following guidelines for the recipe:

Guidelines

- Each use case will be equivalent to a single function

- The use case function will return a union type with two cases: `Success` and `Failure`.
- The use case function will be built from a series of smaller functions, each representing one step in a data flow.
- The errors from each step will be combined into a single error path.

Railway oriented programming

UPDATE: Slides and video from a more comprehensive presentation available here (and if you understand the Either monad, read this first!)

In the previous post, we saw how a use case could be broken into steps, and all the errors shunted off onto a separate error track, like this:

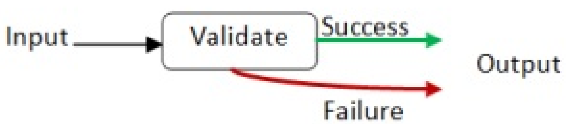


In this post, we'll look at various ways of connecting these step functions into a single unit. The detailed internal design of the functions will be described in a later post.

Designing a function that represents a step

Let's have a closer look at these steps. For example, consider the validation function. How would it work? Some data goes in, but what comes out?

Well, there are two possible cases: either the data is valid (the happy path), or something is wrong, in which case we go onto the failure path and bypass the rest of the steps, like this:



But as before, this would not be a valid function. A function can only have one output, so we must use the `Result` type we defined last time:

```
type Result<'TSuccess,'TFailure> =
    | Success of 'TSuccess
    | Failure of 'TFailure
```

And the diagram now looks like this:



To show you how this works in practice, here is an example of what an actual validation function might look like:

```
type Request = {name:string; email:string}

let validateInput input =
   if input.name = "" then Failure "Name must not be blank"
   else if input.email = "" then Failure "Email must not be blank"
   else Success input  // happy path
```

If you look at the type of the function, the compiler has deduced that it takes a `Request` and spits out a `Result` as output, with a `Request` for the success case and a `string` for the failure case:

```
validateInput : Request -> Result<Request,string>
```

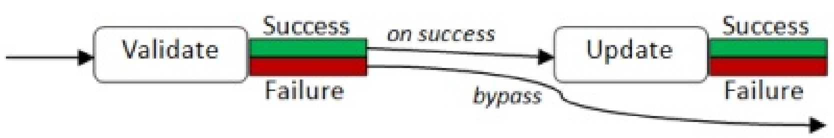
We can analyze the other steps in the flow in the same way. We will find that each one will have the same "shape" -- some sort of input and then this Success/Failure output.

A pre-emptive apology: Having just said that a function can't have two outputs, I may occasionally refer to them hereafter as "two output" functions! Of course, what I mean is that the shape of the function output has two cases.

Railway oriented programming

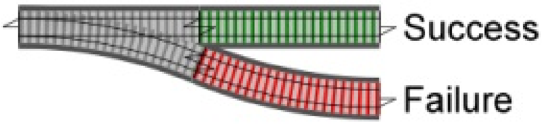
So we have a lot of these "one input -> Success/Failure output" functions -- how do we connect them together?

What we want to do is connect the `Success` output of one to the input of the next, but somehow bypass the second function in case of a `Failure` output. This diagram gives the general idea:

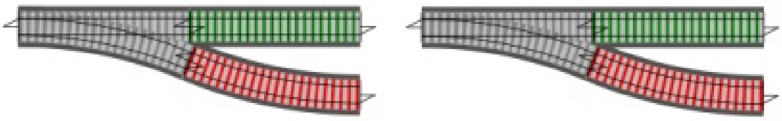


There is a great analogy for doing this -- something you are probably already familiar with. Railways!

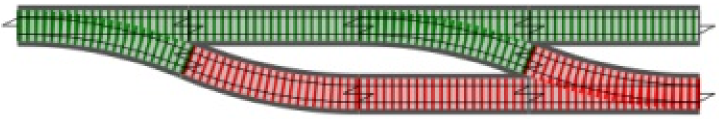
Railways have switches ("points" in the UK) for directing trains onto a different track. We can think of these "Success/Failure" functions as railway switches, like this:



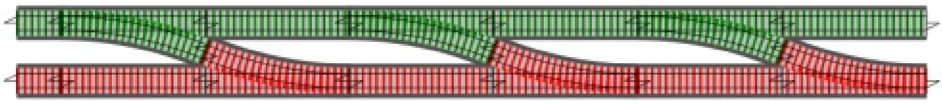
And here we have two in a row.



How do we combine them so that both failure tracks are connected? It's obvious -- like this!



And if we have a whole series of switches, we will end up with a two track system, looking something like this:

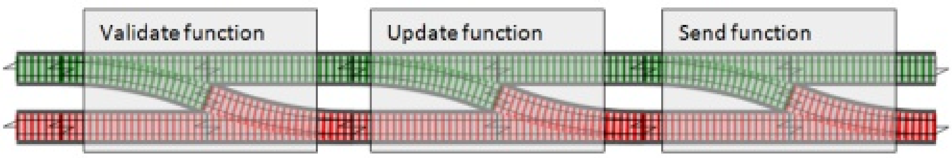


The top track is the happy path, and the bottom track is the failure path.

Now stepping back and looking at the big picture, we can see that we will have a series of black box functions that appear to be straddling a two-track railway, each function processing data and passing it down the track to the next function:



But if we look inside the functions, we can see that there is actually a switch inside each one, for shunting bad data onto the failure track:



Note that once we get on the failure path, we never (normally) get back onto the happy path. We just bypass the rest of the functions until we reach the end.

Basic composition

Before we discuss how to "glue" the step functions together, let's review how composition works.

Imagine that a standard function is a black box (a tunnel, say) sitting on a one-track railway. It has one input and one output.

If we want to connect a series of these one-track functions, we can use the left-to-right composition operator, with the symbol `>>`.



The same composition operation also works with two-track functions as well:



The only constraint on composition is that the output type of the left-hand function has to match the input type of the right-hand function.

In our railway analogy, this means that you can connect one-track output to one-track input, or two-track output to two-track input, but you *can't* directly connect two-track output to one-track input.



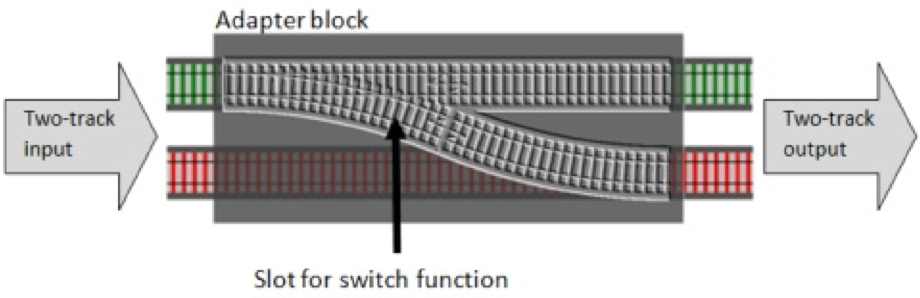
Converting switches to two-track inputs

So now we have run into a problem.

The function for each step is going to be a switch, with *one* input track. But the overall flow requires a *two-track* system, with each function straddling *both* tracks, meaning that each function must have a two-track input (the `Result` output by the previous function), not just a simple one-track input (`Request`).

How can we insert the switches into the two track system?

The answer is simple. We can create an "adapter" function that has a "hole" or "slot" for a switch function and converts it into a proper two-track function. Here's an illustration:



And here's what the actual code looks like. I'm going to name the adapter function `bind`, which is the standard name for it.

```
let bind switchFunction =
    fun twoTrackInput ->
        match twoTrackInput with
        | Success s -> switchFunction s
        | Failure f -> Failure f
```

The bind function takes a switch function as a parameter and returns a new function. The new function takes a two-track input (which is of type `Result`) and then checks each case. If the input is a `Success` it calls the `switchFunction` with the value. But if the input is a `Failure`, then the switch function is bypassed.

Compile it and then look at the function signature:

```
val bind : ('a -> Result<'b,'c>) -> Result<'a,'c> -> Result<'b,'c>
```

One way of interpreting this signature is that the `bind` function has one parameter, a switch function (`'a -> Result<..>`) and it returns a fully two-track function (`Result<..> -> Result<..>`) as output.

To be even more specific:

- The parameter (`switchFunction`) of bind takes some type `'a` and emits a `Result` of type `'b` (for the success track) and `'c` (for the failure track)
- The returned function itself has a parameter (`twoTrackInput`) which is a `Result` of type `'a` (for success) and `'c` (for failure). The type `'a` has to be the same as what the `switchFunction` is expecting on its one track.
- The output of the returned function is another `Result`, this time of type `'b` (for success) and `'c` (for failure) -- the same type as the switch function output.

If you think about it, this type signature is exactly what we would expect.

Note that this function is completely generic -- it will work with *any* switch function and *any* types. All it cares about is the "shape" of the `switchFunction`, not the actual types involved.

Other ways of writing the bind function

Just as an aside, there are some other ways of writing functions like this.

One way is to use an explicit second parameter for the `twoTrackInput` rather than defining an internal function, like this:

```
let bind switchFunction twoTrackInput =
    match twoTrackInput with
    | Success s -> switchFunction s
    | Failure f -> Failure f
```

This is exactly the same as the first definition. And if you are wondering how a two parameter function can be exactly the same as a one parameter function, you need to read the post on currying!

Yet another way of writing it is to replace the `match..with` syntax with the more concise `function` keyword, like this:

```
let bind switchFunction =
    function
    | Success s -> switchFunction s
    | Failure f -> Failure f
```

You might see all three styles in other code, but I personally prefer to use the second style (`let bind switchFunction twoTrackInput =`), because I think that having explicit parameters makes the code more readable for non-experts.

Example: Combining some validation functions

Let's write a little bit of code now, to test the concepts.

Let's start with what we already have defined. `Request`, `Result` and `bind`:

```
type Result<'TSuccess,'TFailure> =
    | Success of 'TSuccess
    | Failure of 'TFailure

type Request = {name:string; email:string}

let bind switchFunction twoTrackInput =
    match twoTrackInput with
    | Success s -> switchFunction s
    | Failure f -> Failure f
```

Next we'll create three validation functions, each of which is a "switch" function, with the goal of combining them into one bigger function:

```
let validate1 input =
   if input.name = "" then Failure "Name must not be blank"
   else Success input

let validate2 input =
   if input.name.Length > 50 then Failure "Name must not be longer than 50 chars"
   else Success input

let validate3 input =
   if input.email = "" then Failure "Email must not be blank"
   else Success input
```

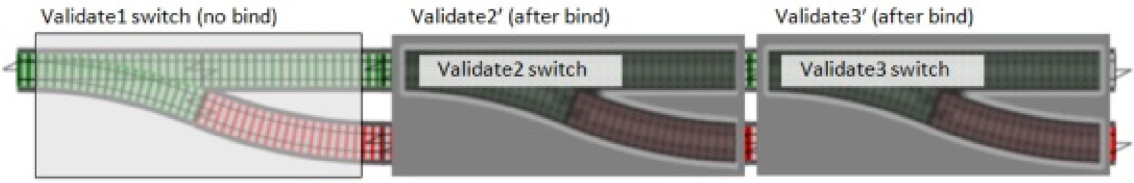
Now to combine them, we apply `bind` to each validation function to create a new alternative function that is two-tracked.

Then we can connect the two-tracked functions using standard function composition, like this:

```
/// glue the three validation functions together
let combinedValidation =
    // convert from switch to two-track input
    let validate2' = bind validate2
    let validate3' = bind validate3
    // connect the two-tracks together
    validate1 >> validate2' >> validate3'
```

The functions `validate2'` and `validate3'` are new functions that take two-track input. If you look at their signatures you will see that they take a `Result` and return a `Result`. But note that `validate1` does not need to be converted to two track input. Its input is left as one-track, and its output is two-track already, as needed for composition to work.

Here's a diagram showing the `Validate1` switch (unbound) and the `Validate2` and `Validate3` switches, together with the `Validate2'` and `Validate3'` adapters.



We could have also "inlined" the `bind`, like this:

```
let combinedValidation =
    // connect the two-tracks together
    validate1
    >> bind validate2
    >> bind validate3
```

Let's test it with two bad inputs and a good input:

```
// test 1
let input1 = {name=""; email=""}
combinedValidation input1
|> printfn "Result1=%A"

// ==> Result1=Failure "Name must not be blank"

// test 2
let input2 = {name="Alice"; email=""}
combinedValidation input2
|> printfn "Result2=%A"

// ==> Result2=Failure "Email must not be blank"

// test 3
let input3 = {name="Alice"; email="good"}
combinedValidation input3
|> printfn "Result3=%A"

// ==> Result3=Success {name = "Alice"; email = "good";}
```

I would encourage you to try it for yourself and play around with the validation functions and test input.

You might be wondering if there is a way to run all three validations in parallel, rather than serially, so that you can get back all the validation errors at once. Yes, there is a way, which I'll explain later in this post.

Bind as a piping operation

While we are discussing the `bind` function, there is a common symbol for it, `>>=`, which is used to pipe values into switch functions.

Here's the definition, which switches around the two parameters to make them easier to chain together:

```
/// create an infix operator
let (>>=) twoTrackInput switchFunction =
    bind switchFunction twoTrackInput
```

One way to remember the symbol is to think of it as the composition symbol, `>>`, followed by a two-track railway symbol, `=`.

When used like this, the `>>=` operator is sort of like a pipe (`|>`) but for switch functions.

In a normal pipe, the left hand side is a one-track value, and the right hand value is a normal function. But in a "bind pipe" operation, the left hand side is a *two-track* value, and the right hand value is a *switch function*.

Here it is in use to create another implementation of the `combinedValidation` function.

```
let combinedValidation x =
    x
    |> validate1   // normal pipe because validate1 has a one-track input
                   // but validate1 results in a two track output...
    >>= validate2  // ... so use "bind pipe". Again the result is a two track output
    >>= validate3   // ... so use "bind pipe" again.
```

The difference between this implementation and the previous one is that this definition is *data-oriented* rather than *function-oriented*. It has an explicit parameter for the initial data value, namely `x`. `x` is passed to the first function, and then the output of that is passed to the second function, and so on.

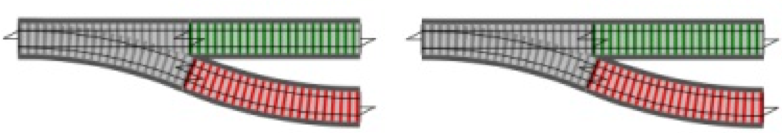
In the previous implementation (repeated below), there was no data parameter at all! The focus was on the functions themselves, not the data that flows through them.

```
let combinedValidation =
    validate1
    >> bind validate2
    >> bind validate3
```

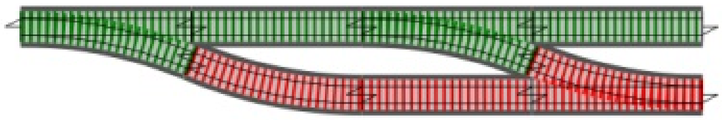
An alternative to bind

Another way to combine switches is not by adapting them to a two track input, but simply by joining them directly together to make a new, bigger switch.

In other words, this:



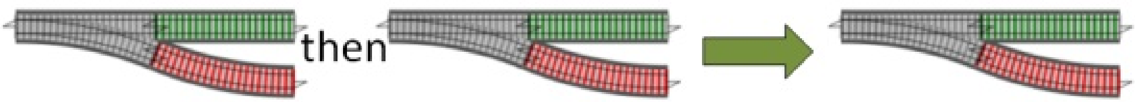
becomes this:



But if you think about it, this combined track is actually just another switch! You can see this if you cover up the middle bit. There's one input and two outputs:



So what we have really done is a form of composition for switches, like this:



Because each composition results in just another switch, we can always add another switch again, resulting in an even bigger thing that is still a switch, and so on.

Here's the code for switch composition. The standard symbol used is `>=>`, a bit like the normal composition symbol, but with a railway track between the angles.

```
let (>=>) switch1 switch2 x =
    match switch1 x with
    | Success s -> switch2 s
    | Failure f -> Failure f
```

Again, the actual implementation is very straightforward. Pass the single track input `x` through the first switch. On success, pass the result into the second switch, otherwise bypass the second switch completely.

Now we can rewrite the `combinedValidation` function to use switch composition rather than bind:

```
let combinedValidation =
    validate1
    >=> validate2
    >=> validate3
```

This one is the simplest yet, I think. It's very easy to extend of course, if we have a fourth validation function, we can just append it to the end.

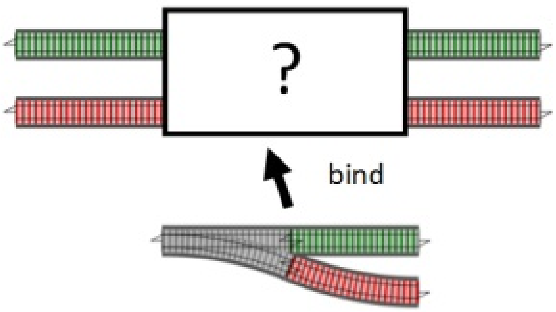
Bind vs. switch composition

We have two different concepts that at first glance seem quite similar. What's the difference?

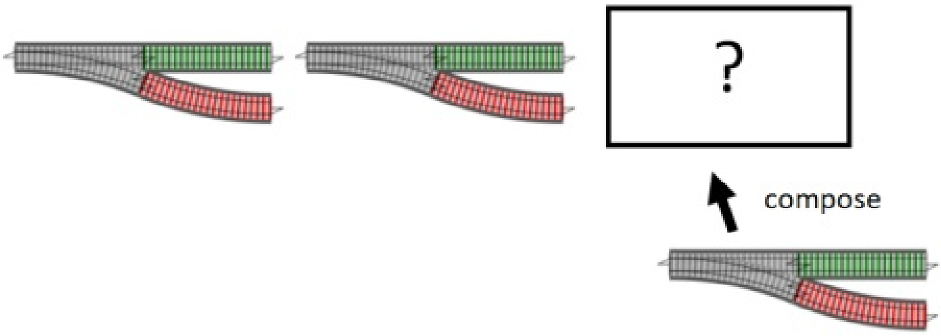
To recap:

- **Bind** has *one* switch function parameter. It is an adapter that converts the switch function into a fully two-track function (with two-track input and two-track output).
- **Switch composition** has *two* switch function parameters. It combines them in series to make another switch function.

So why would you use bind rather than switch composition? It depends on the context. If you have an existing two-track system, and you need to insert a switch, then you have to use bind as an adapter to convert the switch into something that takes two-track input.



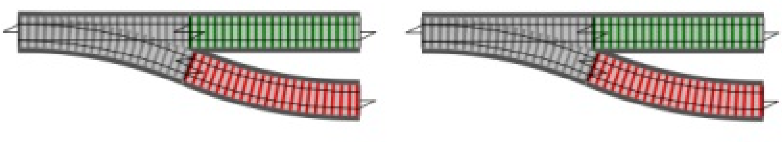
On the other hand, if your entire data flow consists of a chain of switches, then switch composition can be simpler.



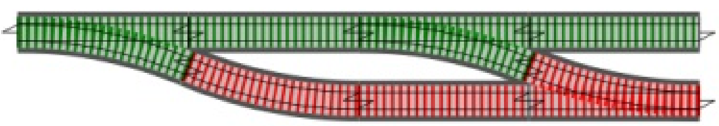
Switch composition in terms of bind

As it happens, switch composition can be written in terms of bind. If you connect the first switch with a bind-adapted second switch, you get the same thing as switch composition:

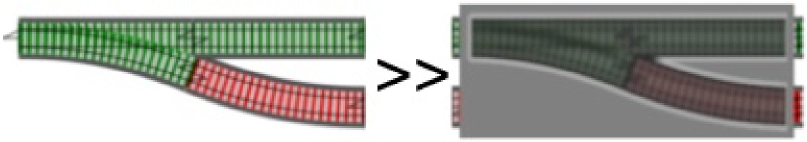
Here are two separate switches:



And then here are the switches composed together to make a new bigger switch:



And here's the same thing done by using `bind` on the second switch:



Here's the switch composition operator rewritten using this way of thinking:

```
let (>=>) switch1 switch2 =
    switch1 >> (bind switch2)
```

This implementation of switch composition is much simpler than the first one, but also more abstract. Whether it is easier to comprehend for a beginner is another matter! I find that if you think of functions as things in their own right, rather than just as conduits for data, this approach becomes easier to understand.

Converting simple functions to the railway-oriented programming model

Once you get the hang of it, you can fit all sorts of other things into this model.

For example, let's say we have a function that is *not* a switch, just a regular function. And say that we want to insert it into our flow.

Here's a real example - say that we want to trim and lowercase the email address after the validation is complete. Here's some code to do this:

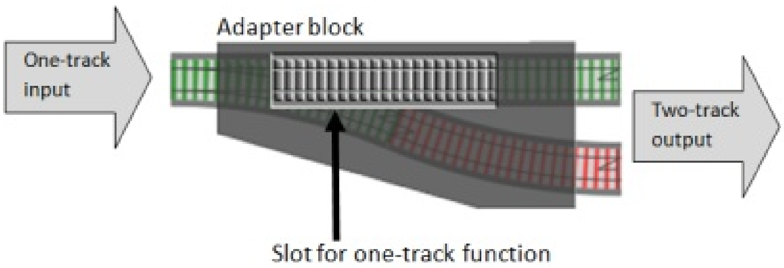
```
let canonicalizeEmail input =
   { input with email = input.email.Trim().ToLower() }
```

This code takes a (single-track) `Request` and returns a (single-track) `Request`.

How can we insert this after the validation steps but before the update step?

Well, if we can turn this simple function into a switch function, then we can use the switch composition we just talked about above.

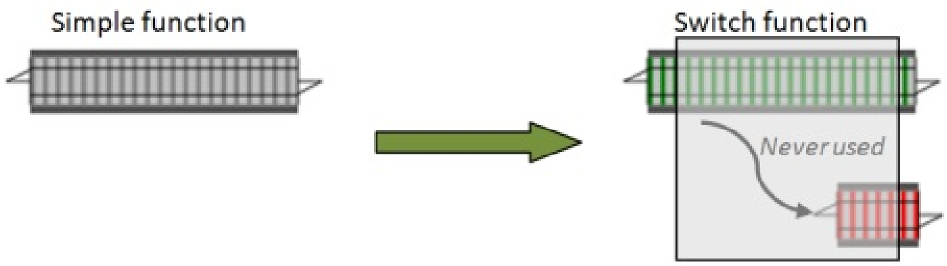
In other words, we need an adapter block. It the same concept that we used for `bind`, except that this time our adapter block will have a slot for one-track function, and the overall "shape" of the adapter block is a switch.



The code to do this is trivial. All we need to do is take the output of the one track function and turn it into a two-track result. In this case, the result will *always* be Success.

```
// convert a normal function into a switch
let switch f x =
    f x |> Success
```

In railway terms, we have added a bit of failure track. Taken as a whole, it *looks* like a switch function (one-track input, two-track output), but of course, the failure track is just a dummy and the switch never actually gets used.



Once `switch` is available, we can easily append the `canonicalizeEmail` function to the end of the chain. Since we are beginning to extend it, let's rename the function to `usecase`.

```
let usecase =
    validate1
    >=> validate2
    >=> validate3
    >=> switch canonicalizeEmail
```

Try testing it to see what happens:

```
let goodInput = {name="Alice"; email="UPPERCASE   "}
usecase goodInput
|> printfn "Canonicalize Good Result = %A"

//Canonicalize Good Result = Success {name = "Alice"; email = "uppercase";}

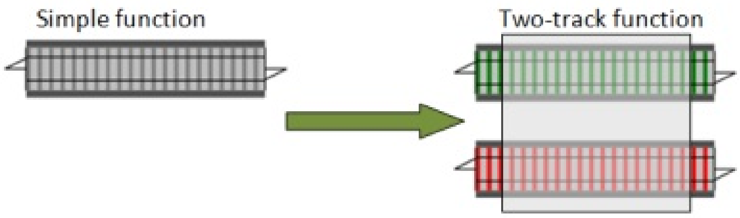
let badInput = {name=""; email="UPPERCASE   "}
usecase badInput
|> printfn "Canonicalize Bad Result = %A"

//Canonicalize Bad Result = Failure "Name must not be blank"
```

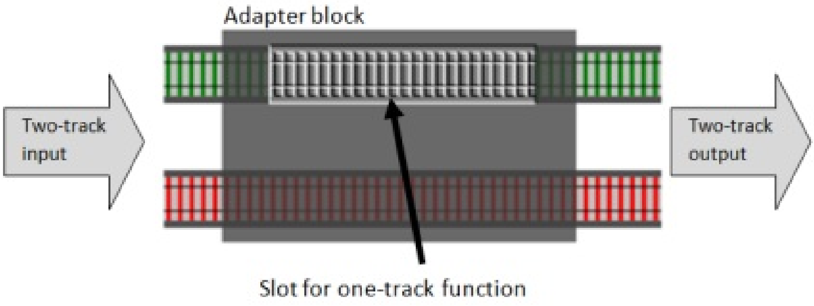
Creating two-track functions from one-track functions

In the previous example, we took a one-track function and created a switch from it. This enabled us to use switch composition with it.

Sometimes though, you want to use the two-track model directly, in which case you want to turn a one-track function into a two-track function directly.



Again, we just need an adapter block with a slot for the simple function. We typically call this adapter `map`.



And again, the actual implementation is very straightforward. If the two-track input is `Success`, call the function, and turn its output into Success. On the other hand, if the two-track input is `Failure` bypass the function completely.

Here's the code:

```
// convert a normal function into a two-track function
let map oneTrackFunction twoTrackInput =
    match twoTrackInput with
    | Success s -> Success (oneTrackFunction s)
    | Failure f -> Failure f
```

And here it is in use with `canonicalizeEmail`:

```
let usecase =
    validate1
    >=> validate2
    >=> validate3
    >> map canonicalizeEmail  // normal composition
```

Note that *normal* composition is now used because `map canonicalizeEmail` is a fully two-track function and can be connected to the output of the `validate3` switch directly.

In other words, for one-track functions, `>=> switch` is exactly the same as `>> map`. Your choice.

Converting dead-end functions to two-track functions

Another function we will often want to work with is a "dead-end" function -- a function that accepts input but has no useful output.

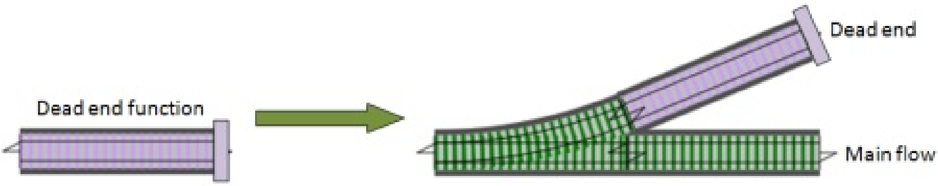
For example, consider a function that updates a database record. It is useful only for its side-effects -- it doesn't normally return anything.

How can we incorporate this kind of function into the flow?

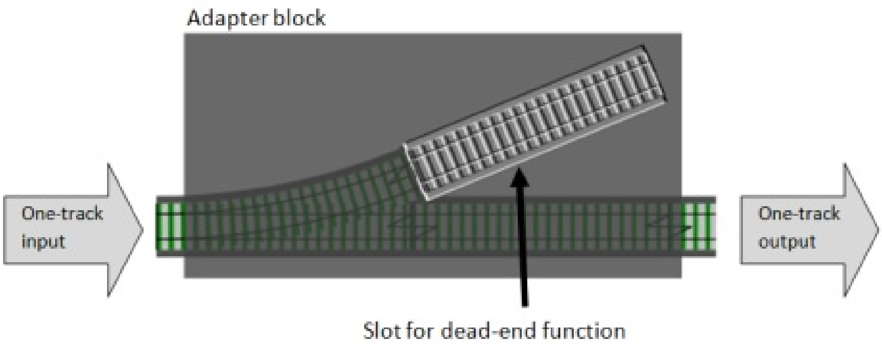
What we need to do is:

- Save a copy of the input.
- Call the function and ignore its output, if any.
- Return the original input for passing on to the next function in the chain.

From a railway point of view, this is equivalent to creating a dead-end siding, like this.



To make this work, we need another adapter function, like `switch`, except that this time it has a slot for one-track dead-end function, and converts it into a single-track pass through function, with a one-track output.



Here's the code, which I will call `tee`, after the UNIX tee command:

```
let tee f x =
    f x |> ignore
    x
```

Once we have converted the dead-end function to a simple one-track pass through function, we can then use it in the data flow by converting it using `switch` or `map` as described above.

Here's the code in use with the "switch composition" style:

```
// a dead-end function
let updateDatabase input =
   ()   // dummy dead-end function for now

let usecase =
    validate1
    >=> validate2
    >=> validate3
    >=> switch canonicalizeEmail
    >=> switch (tee updateDatabase)
```

Or alternatively, rather than using `switch` and then connecting with `>=>`, we can use `map` and connect with `>>`.

Here's a variant implementation which is exactly the same but uses the "two-track" style with normal composition

```
let usecase =
    validate1
    >> bind validate2
    >> bind validate3
    >> map canonicalizeEmail
    >> map (tee updateDatabase)
```

Handling exceptions

Our dead end database update might not return anything, but that doesn't mean that it might not throw an exception. Rather than crashing, we want to catch that exception and turn it into a failure.

The code is similar to the `switch` function, except that it catches exceptions. I'll call it `tryCatch`:

```
let tryCatch f x =
    try
        f x |> Success
    with
    | ex -> Failure ex.Message
```

And here is a modified version of the data flow, using `tryCatch` rather than `switch` for the update database code.

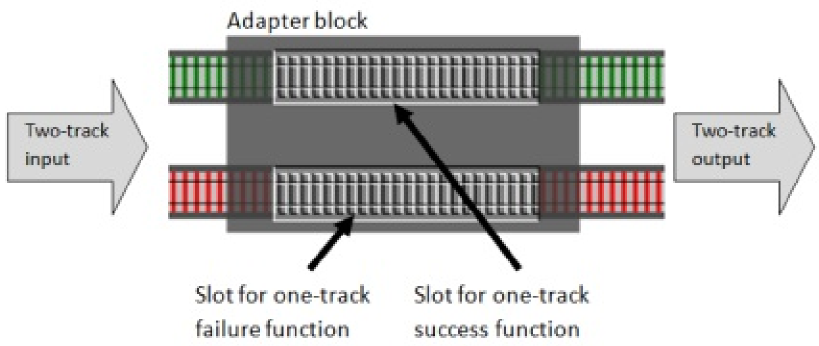
```
let usecase =
    validate1
    >=> validate2
    >=> validate3
    >=> switch canonicalizeEmail
    >=> tryCatch (tee updateDatabase)
```

Functions with two-track input

All the functions we have seen so far have only one input, because they always just work with data travelling along the happy path.

Sometimes though, you *do* need a function that handles both tracks. For example, a logging function that logs errors as well as successes.

As we have done previously, we will create an adapter block, but this time it will have slots for *two* separate one-track functions.



Here's the code:

```
let doubleMap successFunc failureFunc twoTrackInput =
    match twoTrackInput with
    | Success s -> Success (successFunc s)
    | Failure f -> Failure (failureFunc f)
```

As an aside, we can use this function to create a simpler version of `map`, using `id` for the failure function:

```
let map successFunc =
    doubleMap successFunc id
```

Let's use `doubleMap` to insert some logging into the data flow:

```
let log twoTrackInput =
    let success x = printfn "DEBUG. Success so far: %A" x; x
    let failure x = printfn "ERROR. %A" x; x
    doubleMap success failure twoTrackInput

let usecase =
    validate1
    >=> validate2
    >=> validate3
    >=> switch canonicalizeEmail
    >=> tryCatch (tee updateDatabase)
    >> log
```

Here's some test code, with the results:

```
let goodInput = {name="Alice"; email="good"}
usecase goodInput
|> printfn "Good Result = %A"

// DEBUG. Success so far: {name = "Alice"; email = "good";}
// Good Result = Success {name = "Alice"; email = "good";}

let badInput = {name=""; email=""}
usecase badInput
|> printfn "Bad Result = %A"

// ERROR. "Name must not be blank"
// Bad Result = Failure "Name must not be blank"
```

Converting a single value to a two-track value

For completeness, we should also create simple functions that turn a single simple value into a two-track value, either success or failure.

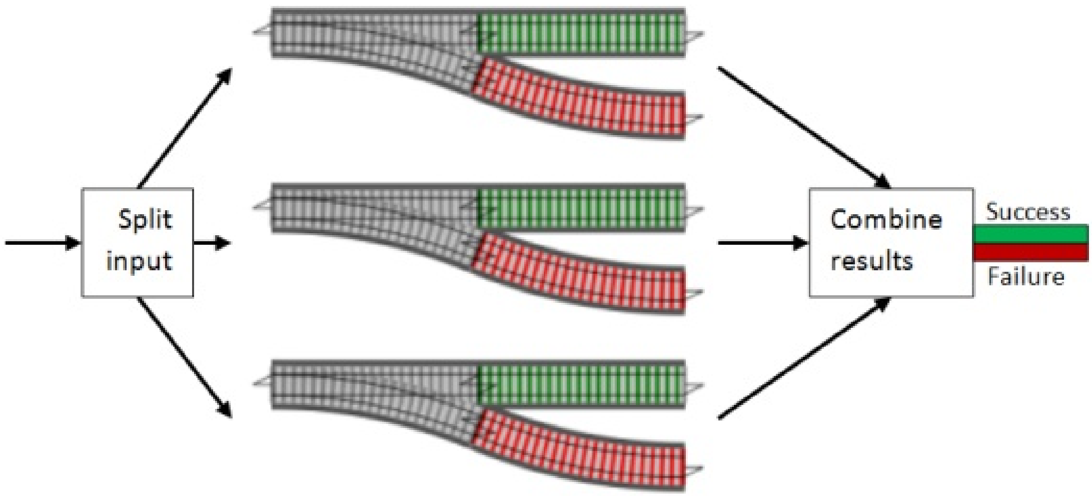
```
let succeed x =
    Success x

let fail x =
    Failure x
```

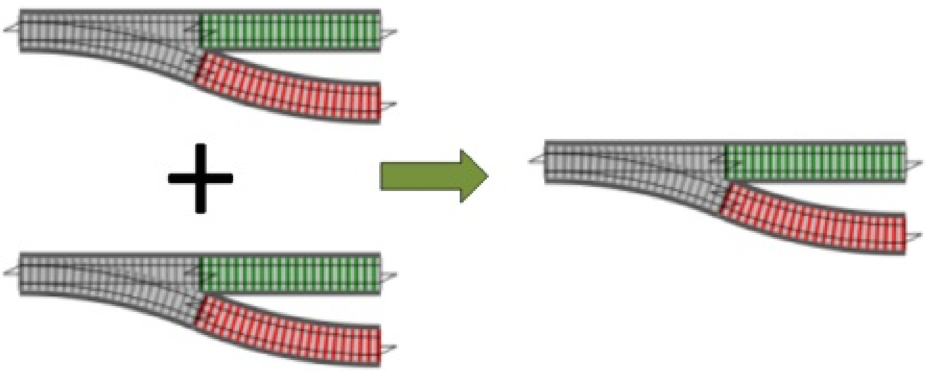
Right now these are trivial, just calling the constructor of the `Result` type, but when we get down to some proper coding we'll see that by using these rather than the union case constructor directly, we can isolate ourselves from changes behind the scenes.

Combining functions in parallel

So far, we have combined functions in series. But with something like validation, we might want to run multiple switches in parallel, and combine the results, like this:



To make this easier, we can reuse the same trick that we did for switch composition. Rather than doing many at once, if we just focus on a single pair, and "add" them to make a new switch, we can then easily chain the "addition" together so that we can add as many as we want. In other words, we just need to implement this:



So, what is the logic for adding two switches in parallel?

- First, take the input and apply it to each switch.
- Next look at the outputs of both switches, and if both are successful, the overall result is `Success` .
- If either output is a failure, then the overall result is `Failure` as well.

Here's the function, which I will call `plus` :

```
let plus switch1 switch2 x =
    match (switch1 x),(switch2 x) with
    | Success s1,Success s2 -> Success (s1 + s2)
    | Failure f1,Success _  -> Failure f1
    | Success _ ,Failure f2 -> Failure f2
    | Failure f1,Failure f2 -> Failure (f1 + f2)
```

But we now have a new problem. What do we do with two successes, or two failures? How do we combine the inner values?

I used `s1 + s2` and `f1 + f2` in the example above, but that implies that there is some sort of `+` operator we can use. That may be true for strings and ints, but it is not true in general.

The method of combining values might change in different contexts, so rather than trying to solve it once and for all, let's punt by letting the caller pass in the functions that are needed.

Here's a rewritten version:

```
let plus addSuccess addFailure switch1 switch2 x =
    match (switch1 x),(switch2 x) with
    | Success s1,Success s2 -> Success (addSuccess s1 s2)
    | Failure f1,Success _  -> Failure f1
    | Success _ ,Failure f2 -> Failure f2
    | Failure f1,Failure f2 -> Failure (addFailure f1 f2)
```

I have put these new functions first in the parameter list, to aid partial application.

An implementation for parallel validation

Now let's create a implementation of "plus" for the validation functions.

- When both functions succeed, they will return the request unchanged, so the `addSuccess` function can return either parameter.
- When both functions fail, they will return different strings, so the `addFailure` function should concatenate them.

For validation then, the "plus" operation that we want is like an "AND" function. Only if both parts are "true" is the result "true".

That naturally leads to wanting to use `&&` as the operator symbol. Unfortunately, `&&` is reserved, but we can use `&&&`, like this:

```
// create a "plus" function for validation functions
let (&&&) v1 v2 =
    let addSuccess r1 r2 = r1 // return first
    let addFailure s1 s2 = s1 + "; " + s2  // concat
    plus addSuccess addFailure v1 v2
```

And now using `&&&`, we can create a single validation function that combines the three smaller validations:

```
let combinedValidation =
    validate1
    &&& validate2
    &&& validate3
```

Now let's try it with the same tests we had earlier:

```
// test 1
let input1 = {name=""; email=""}
combinedValidation input1
|> printfn "Result1=%A"
// ==>  Result1=Failure "Name must not be blank; Email must not be blank"

// test 2
let input2 = {name="Alice"; email=""}
combinedValidation input2
|> printfn "Result2=%A"
// ==>  Result2=Failure "Email must not be blank"

// test 3
let input3 = {name="Alice"; email="good"}
combinedValidation input3
|> printfn "Result3=%A"
// ==>  Result3=Success {name = "Alice"; email = "good";}
```

The first test now has *two* validation errors combined into a single string, just as we wanted.

Next, we can tidy up the main dataflow function by using the `usecase` function now instead of the three separate validation functions we had before:

```
let usecase =
    combinedValidation
    >=> switch canonicalizeEmail
    >=> tryCatch (tee updateDatabase)
```

And if we test that now, we can see that a success flows all the way to the end and that the email is lowercased and trimmed:

```
// test 4
let input4 = {name="Alice"; email="UPPERCASE   "}
usecase input4
|> printfn "Result4=%A"
// ==>  Result4=Success {name = "Alice"; email = "uppercase";}
```

You might be asking, can we create a way of OR-ing validation functions as well? That is, the overall result is valid if either part is valid? The answer is yes, of course. Try it! I suggest that you use the symbol `|||` for this.

Dynamic injection of functions

Another thing we might want to do is add or remove functions into the flow dynamically, based on configuration settings, or even the content of the data.

The simplest way to do this is to create a two-track function to be injected into the stream, and replace it with the `id` function if not needed.

Here's the idea:

```
let injectableFunction =
    if config.debug then debugLogger else id
```

Let's try it with some real code:

```
type Config = {debug:bool}

let debugLogger twoTrackInput =
    let success x = printfn "DEBUG. Success so far: %A" x; x
    let failure = id // don't log here
    doubleMap success failure twoTrackInput

let injectableLogger config =
    if config.debug then debugLogger else id

let usecase config =
    combinedValidation
    >> map canonicalizeEmail
    >> injectableLogger config
```

And here is it in use:

```
let input = {name="Alice"; email="good"}

let releaseConfig = {debug=false}
input
|> usecase releaseConfig
|> ignore

// no output

let debugConfig = {debug=true}
input
|> usecase debugConfig
|> ignore

// debug output
// DEBUG. Success so far: {name = "Alice"; email = "good";}
```

The railway track functions: A toolkit

Let's step back and review what we have done so far.

Using railway track as a metaphor, we have created a number of useful building blocks that will work with *any* data-flow style application.

We can classify our functions roughly like this:

- **"constructors"** are used to create new track.
- **"adapters"** convert one kind of track into another.
- **"combiners"** link sections of track together to make a bigger piece of track.

These functions form what can be loosely called a *combinator library*, that is, a group of functions that are designed to work with a type (here represented by railway track), with the design goal that bigger pieces can be built by adapting and combining smaller pieces.

Functions like `bind` , `map` , `plus` , etc., crop up in all sorts of functional programming scenarios, and so you can think of them as functional patterns -- similar to, but not the same as, the OO patterns such as "visitor", "singleton", "facade", etc.

Here they all are together:

Concept	Description
`succeed`	A constructor that takes a one-track value and creates a two-track value on the Success branch. In other contexts, this might also be called `return` or `pure`.
`fail`	A constructor that takes a one-track value and creates a two-track value on the Failure branch.
`bind`	An adapter that takes a switch function and creates a new function that accepts two-track values as input.
`>>=`	An infix version of bind for piping two-track values into switch functions.
`>>`	Normal composition. A combiner that takes two normal functions and creates a new function by connecting them in series.
`>=>`	Switch composition. A combiner that takes two switch functions and creates a new switch function by connecting them in series.
`switch`	An adapter that takes a normal one-track function and turns it into a switch function. (Also known as a "lift" in some contexts.)
`map`	An adapter that takes a normal one-track function and turns it into a two-track function. (Also known as a "lift" in some contexts.)
`tee`	An adapter that takes a dead-end function and turns it into a one-track function that can be used in a data flow. (Also known as `tap`.)
`tryCatch`	An adapter that takes a normal one-track function and turns it into a switch function, but also catches exceptions.
`doubleMap`	An adapter that takes two one-track functions and turns them into a single two-track function. (Also known as `bimap`.)
`plus`	A combiner that takes two switch functions and creates a new switch function by joining them in "parallel" and "adding" the results. (Also known as `++` and in other contexts.)
`&&&`	The "plus" combiner tweaked specifically for the validation functions, modelled on a binary AND.

The railway track functions: complete code

Here is the complete code for all the functions in one place.

I have made some minor tweaks from the original code presented above:

- Most functions are now defined in terms of a core function called `either`.
- `tryCatch` has been given an extra parameter for the exception handler.

```
// the two-track type
type Result<'TSuccess,'TFailure> =
    | Success of 'TSuccess
```

```
    | Failure of 'TFailure

// convert a single value into a two-track result
let succeed x =
    Success x

// convert a single value into a two-track result
let fail x =
    Failure x

// apply either a success function or failure function
let either successFunc failureFunc twoTrackInput =
    match twoTrackInput with
    | Success s -> successFunc s
    | Failure f -> failureFunc f

// convert a switch function into a two-track function
let bind f =
    either f fail

// pipe a two-track value into a switch function
let (>>=) x f =
    bind f x

// compose two switches into another switch
let (>=>) s1 s2 =
    s1 >> bind s2

// convert a one-track function into a switch
let switch f =
    f >> succeed

// convert a one-track function into a two-track function
let map f =
    either (f >> succeed) fail

// convert a dead-end function into a one-track function
let tee f x =
    f x; x

// convert a one-track function into a switch with exception handling
let tryCatch f exnHandler x =
    try
        f x |> succeed
    with
    | ex -> exnHandler ex |> fail

// convert two one-track functions into a two-track function
let doubleMap successFunc failureFunc =
    either (successFunc >> succeed) (failureFunc >> fail)

// add two switches in parallel
let plus addSuccess addFailure switch1 switch2 x =
```

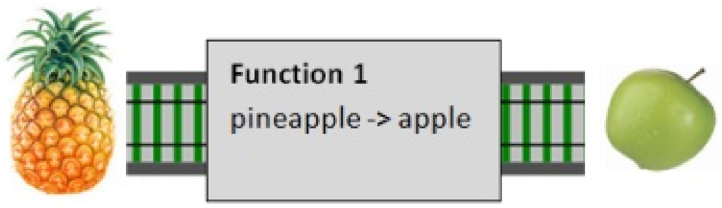
```
    match (switch1 x),(switch2 x) with
    | Success s1,Success s2 -> Success (addSuccess s1 s2)
    | Failure f1,Success _  -> Failure f1
    | Success _ ,Failure f2 -> Failure f2
    | Failure f1,Failure f2 -> Failure (addFailure f1 f2)
```

Types vs. shapes

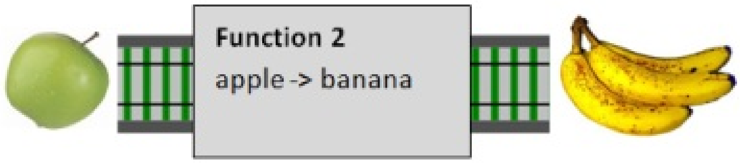
So far, we have focused entirely on the shape of the track, not the cargo on the trains.

This is a magical railway, where the goods being carried can change as they go along each length of track.

For example, a cargo of pineapples will magically transform into apples when it goes through the tunnel called `function1`.



And a cargo of apples will transform into bananas when it goes through the tunnel called `function2`.



This magical railway has an important rule, namely that you can only connect tracks which carry the same type of cargo. In this case we *can* connect `function1` to `function2` because the cargo coming out of `function1` (apples) is the same as the cargo going into `function2` (also apples).



Of course, it is not always true that the tracks carry the same cargo, and a mismatch in the kind of cargo will cause an error.

But you'll notice that in this discussion so far, we haven't mentioned the cargo once! Instead, we have spent all our time talking about one-track vs. two track functions.

Of course, it goes without saying that the cargo must match up. But I hope you can see that it is the *shape* of the track that is really the important thing, not the cargo that is carried.

Generic types are powerful

Why have we not worried about the type of cargo? Because all the "adapter" and "combiner" functions are completely generic! The `bind` and `map` and `switch` and `plus` functions do not care about the type of the cargo, only the *shape* of the track.

Having extremely generic functions is a benefit in two ways. The first way is obvious: the more generic a function is, the more reusable it is. The implementation of `bind` will work with any types (as long as the shape is right).

But there is another, more subtle aspect of generic functions that is worth pointing out. Because we generally know *nothing* about the types involved, we are very constrained in what we can and can't do. As a result, we can't introduce bugs!

To see what I mean, let's look at the signature for `map` :

```
val map : ('a -> 'b) -> (Result<'a,'c> -> Result<'b,'c>)
```

It takes a function parameter `'a -> 'b` and a value `Result<'a,'c>` and returns a value `Result<'b,'c>` .

We don't know anything about the types `'a` , `'b` , and `'c` . The only things we know are that:

- The *same* type `'a` shows up in both the function parameter and the `Success` case of the first `Result` .
- The *same* type `'b` shows up in both the function parameter and the `Success` case of the second `Result` .
- The *same* type `'c` shows up in the `Failure` cases of both the first and second `Result` s, but doesn't show up in the function parameter at all.

What can we deduce from this?

The return value has a type `'b` in it. But where does it come from? We don't know what type `'b` is, so we don't know how to make one. But the function parameter knows how to make one! Give it an `'a` and it will make a `'b` for us.

But where can we get an `'a` from? We don't know what type `'a` is either, so again we don't know how to make one. But the first result parameter has an `'a` we can use, so you can see that we are *forced* to get the `Success` value from the `Result<'a,'c>` parameter and

pass it to the function parameter. And then the `Success` case of the `Result<'b,'c>` return value *must* be constructed from the result of the function.

Finally, the same logic applies to `'c`. We are forced to get the `Failure` value from the `Result<'a,'c>` input parameter and use it to construct the `Failure` case of the `Result<'a,'c>` return value.

In other words, there is basically *only one way to implement the `map` function*! The type signature is so generic that we have no choice.

On the other hand, imagine that the `map` function had been very specific about the types it needed, like this:

```
val map : (int -> int) -> (Result<int,int> -> Result<int,int>)
```

In this case, we can come up a huge number of different implementations. To list a few:

- We could have swapped the success and failure tracks.
- We could have added a random number to the success track.
- We could have ignored the function parameter altogether, and returned zero on both the success and failure tracks.

All of these implementations are "buggy" in the sense that they don't do what we expect. But they are all only possible because we know in advance that the type is `int`, and therefore we can manipulate the values in ways we are not supposed to. The less we know about the types, the less likely we are to make a mistake.

The failure type

In most of our functions, the transformation only applies to the success track. The failure track is left alone (`map`), or merged with an incoming failure (`bind`).

This implies that the failure track must be *same type* all the way through. In this post we have just used `string`, but in the next post we'll change the failure type to be something more useful.

Summary and guidelines

At the beginning of this series, I promised to give you a simple recipe that you could follow.

But you might be feeling a bit overwhelmed now. Instead of making things simpler, I seem to have made things more complicated. I have shown you lots of different ways of doing the same thing! Bind vs. compose. Map vs. switch. Which approach should you use? Which way

is best?

Of course, there is never one "right way" for all scenarios, but nevertheless, as promised, here are some guidelines that can be used as the basis of a reliable and repeatable recipe.

Guidelines

- Use double-track railway as your underlying model for dataflow situations.
- Create a function for each step in the use case. The function for each step can in turn be built from smaller functions (e.g. the validation functions).
- Use standard composition (`>>`) to connect the functions.
- If you need to insert a switch into the flow, use `bind` .
- If you need to insert a single-track function into the flow, use `map` .
- If you need to insert other types of functions into the flow, create an appropriate adapter block and use it.

These guidelines may result in code that is not particularly concise or elegant, but on the other hand, you will be using a consistent model, and it should be understandable to other people when it needs to be maintained.

So with these guidelines, here are the main bits of the implementation so far. Note especially the use of `>>` everywhere in the final `usecase` function.

```
open RailwayCombinatorModule

let (&&&) v1 v2 =
    let addSuccess r1 r2 = r1 // return first
    let addFailure s1 s2 = s1 + "; " + s2  // concat
    plus addSuccess addFailure v1 v2

let combinedValidation =
    validate1
    &&& validate2
    &&& validate3

let canonicalizeEmail input =
   { input with email = input.email.Trim().ToLower() }

let updateDatabase input =
   ()   // dummy dead-end function for now

// new function to handle exceptions
let updateDatebaseStep =
    tryCatch (tee updateDatabase) (fun ex -> ex.Message)

let usecase =
    combinedValidation
    >> map canonicalizeEmail
    >> bind updateDatebaseStep
    >> log
```

One final suggestion. If you are working with a team of non-experts, unfamiliar operator symbols will put people off. So here some extra guidelines with respect to operators:

- Don't use any "strange" operators other than `>>` and `|>`.
- In particular, that means you should *not* use operators like `>>=` or `>=>` unless everyone is aware of them.
- An exception can be made if you define the operator at the top of the module or function where it is used. For example, the `&&&` operator could be defined at the top of the validation module and then used later in that same module.

Further reading

- If you like this "railway oriented" approach, you can also see it applied to FizzBuzz.
- I also have some slides and video that show how take this approach further. (At some point I will turn these into a proper blog post)

I presented on this topic at NDC Oslo 2014 (click image to view video)

And here are the slides I used:

Railway Oriented Programming from **my slideshare page**

Railway oriented programming: Carbonated edition

As a follow up to the Railway Oriented Programming post, I thought I'd apply the same technique to the FizzBuzz problem, and compare it with other implementations.

A large part of this post was directly ~~stolen from~~ inspired by Dave Fayram's post on FizzBuzz, with some additional ideas from raganwald.

FizzBuzz: The imperative version

As a reminder, here are the requirements for the FizzBuzz problem:

```
Write a program that prints the numbers from 1 to 100.
* For multiples of three print "Fizz" instead of the number.
* For multiples of five print "Buzz".
* For numbers which are multiples of both three and five print "FizzBuzz".
```

And here is a basic F# solution:

```
module FizzBuzz_Match =

    let fizzBuzz i =
        match i with
        | _ when i % 15 = 0 ->
            printf "FizzBuzz"
        | _ when i % 3 = 0 ->
            printf "Fizz"
        | _ when i % 5 = 0 ->
            printf "Buzz"
        | _ ->
            printf "%i" i

        printf "; "

    // do the fizzbuzz
    [1..100] |> List.iter fizzBuzz
```

I have defined a function `fizzBuzz` that, given an integer `i` , uses `match` with `when` clauses to do the various tests, and then prints the appropriate value.

Simple and straightforward, and fine for a quick hack, but there are a number of problems with this implementation.

First, note that we had to have a special case for "fifteen". We couldn't just reuse the code from the "three" and "five" cases. And this means that if we want to add another case, such as "seven", we also need to add special cases for all the combinations as well (that is "21", "35" and "105"). And of course, adding more numbers would lead to a combinatorial explosion of cases.

Second, the order of matching is important. If the "fifteen" case had come last in the list of patterns, the code would have run correctly, but not actually met the requirements. And again, if we need to add new cases, we must always remember to put the largest ones first to ensure correctness. This is just the kind of thing that causes subtle bugs.

Let's try another implementation, where we reuse the code for the "three" and "five" case, and eliminate the need for a "fifteen" case altogether:

```
module FizzBuzz_IfPrime =

    let fizzBuzz i =
        let mutable printed = false

        if i % 3 = 0 then
            printed <- true
            printf "Fizz"

        if i % 5 = 0 then
            printed <- true
            printf "Buzz"

        if not printed then
            printf "%i" i

        printf "; "

    // do the fizzbuzz
    [1..100] |> List.iter fizzBuzz
```

In this implementation, the printed value for "15" will be correct, because both the "3" and "5" cases will be used. And also we don't have to worry about order -- as much, anyway.

But -- these branches are no longer independent, so we have to track whether *any* branch has been used at all, so that we can handle the default case. And that has led to the mutable variable. Mutables are a code smell in F#, so this implementation is not ideal.

However, this version *does* have the advantage that it can be easily refactored to support multiple factors, not just 3 and 5.

Below is a version that does just this. We pass in a list of "rules" to `fizzBuzz`. Each rule consists of a factor and a corresponding label to print out. The `fizzBuzz` function then just iterates through these rules, processing each in turn.

```
module FizzBuzz_UsingFactorRules =

    let fizzBuzz rules i  =
        let mutable printed = false

        for factor,label in rules do
            if i % factor = 0 then
                printed <- true
                printf "%s" label

        if not printed then
            printf "%i" i

        printf "; "

    // do the fizzbuzz
    let rules = [ (3,"Fizz"); (5,"Buzz") ]
    [1..100] |> List.iter (fizzBuzz rules)
```

If we want additional numbers to be processed, we just add them to the list of rules, like this:

```
module FizzBuzz_UsingFactorRules =

    // existing code as above

    let rules2 = [ (3,"Fizz"); (5,"Buzz"); (7,"Baz") ]
    [1..105] |> List.iter (fizzBuzz rules2)
```

To sum up, we have created a very imperative implementation that would be almost the same in C#. It's flexible, but the mutable variable is a bit of a code smell. Is there another way?

FizzBuzz: The pipeline version

In this next version, we'll look at using the "pipeline" model, where data is fed through a series of functions to arrive at a final result.

In this design, I envision a pipeline of functions, one to handle the "three" case, one to handle the "five" case, and so on. And at the end, the appropriate label is spat out, ready to be printed.

Here is some pseudocode to demonstrate the concept:

```
data |> handleThreeCase |> handleFiveCase |> handleAllOtherCases |> printResult
```

As an additional requirement, we want the pipeline to have *no* side effects. This means that the intermediate functions must *not* print anything. They must instead pass any generated labels down the pipe to the end, and only at that point print the results.

Designing the pipeline

As a first step, we need to define what data will be fed down the pipe.

Let's start with the first function, called `handleThreeCase` in the pseudocode above. What is its input, and what is its output?

Obviously, the input is the integer being processed. But the output could be the string "Fizz" if we're lucky. Or the original integer if we're not.

So now let's think about the input to the second function, `handleFiveCase`. It needs the integer as well. But in the case of "15" it *also* needs the string "Fizz" as well, so it can append "Buzz" to it.

Finally, the `handleAllOtherCases` function converts the int to a string, but *only* if "Fizz" or "Buzz" have not been generated yet.

It's quite obvious then, that the data structure needs to contain both the integer being processed *and* the "label so far".

The next question is: how do we know if an upstream function has created a label? The `handleAllOtherCases` needs to know this in order to determine whether it needs to do anything.

One way would be to use an empty string (or, horrors, a null string), but let's be good and use a `string option`.

So, here's the final data type that we will be using:

```
type Data = {i:int; label:string option}
```

Pipeline version 1

With this data structure, we can define how `handleThreeCase` and `handleFiveCase` will work.

- First, test the input int `i` to see if it is divisible by the factor.

- If it is divisible, look at the `label` -- if it is `None`, then replace it with `Some "Fizz"` or `Some "Buzz"`.
- If the label already has a value, then append "Buzz" (or whatever) to it.
- If the input is not divisible by the factor, just pass on the data unchanged.

Given this design, here's the implementation. It's a generic function that I will call `carbonate` (after raganwald) because it works with both "Fizz" and "Buzz":

```
let carbonate factor label data =
    let {i=i; label=labelSoFar} = data
    if i % factor = 0 then
        // pass on a new data record
        let newLabel =
            match labelSoFar with
            | Some s -> s + label
            | None -> label
        {data with label=Some newLabel}
    else
        // pass on the unchanged data
        data
```

The design for the `handleAllOtherCases` function is slightly different:

- Look at the label -- if it is not `None`, then a previous function has created a label, so do nothing.
- But if the label is `None`, replace it with the string representation of the integer.

Here's the code -- I will call it `labelOrDefault`:

```
let labelOrDefault data =
    let {i=i; label=labelSoFar} = data
    match labelSoFar with
    | Some s -> s
    | None -> sprintf "%i" i
```

Now that we have the components, we can assemble the pipeline:

```
let fizzBuzz i =
    {i=i; label=None}
    |> carbonate 3 "Fizz"
    |> carbonate 5 "Buzz"
    |> labelOrDefault     // convert to string
    |> printf "%s; "      // print
```

Note that we have to create an initial record using `{i=i; label=None}` for passing into the first function (`carbonate 3 "Fizz"`).

Finally, here is all the code put together:

```
module FizzBuzz_Pipeline_WithRecord =

    type Data = {i:int; label:string option}

    let carbonate factor label data =
        let {i=i; label=labelSoFar} = data
        if i % factor = 0 then
            // pass on a new data record
            let newLabel =
                match labelSoFar with
                | Some s -> s + label
                | None -> label
            {data with label=Some newLabel}
        else
            // pass on the unchanged data
            data

    let labelOrDefault data =
        let {i=i; label=labelSoFar} = data
        match labelSoFar with
        | Some s -> s
        | None -> sprintf "%i" i

    let fizzBuzz i =
        {i=i; label=None}
        |> carbonate 3 "Fizz"
        |> carbonate 5 "Buzz"
        |> labelOrDefault     // convert to string
        |> printf "%s; "      // print

    [1..100] |> List.iter fizzBuzz
```

Pipeline version 2

Creating a new record type can be useful as a form of documentation, but in a case like this, it would probably be more idiomatic to just use a tuple rather than creating a special data structure.

So here's a modified implementation that uses tuples.

```
module FizzBuzz_Pipeline_WithTuple =

    // type Data = int * string option

    let carbonate factor label data =
        let (i,labelSoFar) = data
        if i % factor = 0 then
            // pass on a new data record
            let newLabel =
                labelSoFar
                |> Option.map (fun s -> s + label)
                |> defaultArg <| label
            (i,Some newLabel)
        else
            // pass on the unchanged data
            data

    let labelOrDefault data =
        let (i,labelSoFar) = data
        labelSoFar
        |> defaultArg <| sprintf "%i" i

    let fizzBuzz i =
        (i,None)   // use tuple instead of record
        |> carbonate 3 "Fizz"
        |> carbonate 5 "Buzz"
        |> labelOrDefault     // convert to string
        |> printf "%s; "      // print

    [1..100] |> List.iter fizzBuzz
```

As an exercise, try to find all the code that had to be changed.

Eliminating explicit tests for Some and None

In the tuple code above, I have also replaced the explicit Option matching code `match .. Some .. None` with some built-in Option functions, `map` and `defaultArg`.

Here are the changes in `carbonate` :

```
// before
let newLabel =
    match labelSoFar with
    | Some s -> s + label
    | None -> label

// after
let newLabel =
    labelSoFar
    |> Option.map (fun s -> s + label)
    |> defaultArg <| label
```

and in `labelOrDefault` :

```
// before
match labelSoFar with
| Some s -> s
| None -> sprintf "%i" i

// after
labelSoFar
|> defaultArg <| sprintf "%i" i
```

You might be wondering about the strange looking `|> defaultArg <|` idiom.

I am using it because the option is the *first* parameter to `defaultArg` , not the *second*, so a normal partial application won't work. But "bi-directional" piping does work, hence the strange looking code.

Here's what I mean:

```
// OK - normal usage
defaultArg myOption defaultValue

// ERROR: piping doesn't work
myOption |> defaultArg defaultValue

// OK - bi-directional piping does work
myOption |> defaultArg <| defaultValue
```

Pipeline version 3

Our `carbonate` function is generic for any factor, so we can easily extend the code to support "rules" just as in the earlier imperative version.

But one issue seems to be that we have hard-coded the "3" and "5" cases into the pipeline, like this:

```
|> carbonate 3 "Fizz"
|> carbonate 5 "Buzz"
```

How can we dynamically add new functions into the pipeline?

The answer is quite simple. We dynamically create a function for each rule, and then combine all these functions into one using composition.

Here's a snippet to demonstrate:

```
let allRules =
    rules
    |> List.map (fun (factor,label) -> carbonate factor label)
    |> List.reduce (>>)
```

Each rule is mapped into a function. And then the list of functions is combined into one single function using `>>`.

Putting it all together, we have this final implementation:

```
module FizzBuzz_Pipeline_WithRules =

    let carbonate factor label data =
        let (i,labelSoFar) = data
        if i % factor = 0 then
            // pass on a new data record
            let newLabel =
                labelSoFar
                |> Option.map (fun s -> s + label)
                |> defaultArg <| label
            (i,Some newLabel)
        else
            // pass on the unchanged data
            data

    let labelOrDefault data =
        let (i,labelSoFar) = data
        labelSoFar
        |> defaultArg <| sprintf "%i" i

    let fizzBuzz rules i =

        // create a single function from all the rules
        let allRules =
            rules
            |> List.map (fun (factor,label) -> carbonate factor label)
            |> List.reduce (>>)

        (i,None)
        |> allRules
        |> labelOrDefault     // convert to string
        |> printf "%s; "      // print

    // test
    let rules = [ (3,"Fizz"); (5,"Buzz"); (7,"Baz") ]
    [1..105] |> List.iter (fizzBuzz rules)
```

Comparing this "pipeline" version with the previous imperative version, the design is much more functional. There are no mutables and there are no side-effects anywhere (except in the final `printf` statement).

There is a subtle bug in the use of `List.reduce`, however. Can you see what it is?** For a discussion of the problem and the fix, see the postscript at the bottom of this page.

** Hint: try an empty list of rules.

FizzBuzz: The railway oriented version

The pipeline version is a perfectly adequate functional implementation of FizzBuzz, but for fun, let's see if we can use the "two-track" design described in the railway oriented programming post.

As a quick reminder, in "railway oriented programming" (a.k.a the "Either" monad), we define a union type with two cases: "Success" and "Failure", each representing a different "track". We then connect a set of "two-track" functions together to make the railway.

Most of the functions we actually use are what I called "switch" or "points" functions, with a *one* track input, but a two-track output, one for the Success case, and one for the Failure case. These switch functions are converted into two-track functions using a glue function called "bind".

Here is a module containing definitions of the functions we will need.

```
module RailwayCombinatorModule =

    let (|Success|Failure|) =
        function
        | Choice1Of2 s -> Success s
        | Choice2Of2 f -> Failure f

    /// convert a single value into a two-track result
    let succeed x = Choice1Of2 x

    /// convert a single value into a two-track result
    let fail x = Choice2Of2 x

    // appy either a success function or failure function
    let either successFunc failureFunc twoTrackInput =
        match twoTrackInput with
        | Success s -> successFunc s
        | Failure f -> failureFunc f

    // convert a switch function into a two-track function
    let bind f =
        either f fail
```

I am using the `Choice` type here, which is built into the F# core library. But I have created some helpers to make it look like the Success/Failure type: an active pattern and two constructors.

Now, how will we adapt FizzBuzz to this?

Let's start by doing the obvious thing: defining "carbonation" as success, and an unmatched integer as a failure.

In other words, the Success track contains the labels, and the Failure track contains the ints.

Our `carbonate` "switch" function will therefore look like this:

```
let carbonate factor label i =
    if i % factor = 0 then
        succeed label
    else
        fail i
```

This implementation is similar to the one used in the pipeline design discussed above, but it is cleaner because the input is just an int, not a record or a tuple.

Next, we need to connect the components together. The logic will be:

- if the int is already carbonated, ignore it
- if the int is not carbonated, connect it to the input of the next switch function

Here is the implementation:

```
let connect f =
    function
    | Success x -> succeed x
    | Failure i -> f i
```

Another way of writing this is to use the `either` function we defined in the library module:

```
let connect' f =
    either succeed f
```

Make sure you understand that both of these implementations do exactly the same thing!

Next, we can create our "two-track" pipeline, like this:

```
let fizzBuzz =
    carbonate 15 "FizzBuzz"      // need the 15-FizzBuzz rule because of short-circuit
    >> connect (carbonate 3 "Fizz")
    >> connect (carbonate 5 "Buzz")
    >> either (printf "%s; ") (printf "%i; ")
```

This is superficially similar to the "one-track" pipeline, but actually uses a different technique. The switches are connected together through composition (`>>`) rather than piping (`|>`).

As a result, the `fizzBuzz` function does not have an int parameter -- we are defining a function by combining other functions. There is no data anywhere.

A few other things have changed as well:

- We have had to reintroduce the explicit test for the "15" case. This is because we have only two tracks (success or failure). There is no "half-finished track" that allows the "5" case to add to the output of the "3" case.
- The `labelOrDefault` function from the previous example has been replaced with `either`. In the Success case, a string is printed. In the Failure case, an int is printed.

Here is the complete implementation:

```
module FizzBuzz_RailwayOriented_CarbonationIsSuccess =

    open RailwayCombinatorModule

    // carbonate a value
    let carbonate factor label i =
        if i % factor = 0 then
            succeed label
        else
            fail i

    let connect f =
        function
        | Success x -> succeed x
        | Failure i -> f i

    let connect' f =
        either succeed f

    let fizzBuzz =
        carbonate 15 "FizzBuzz"      // need the 15-FizzBuzz rule because of short-cir
cuit
        >> connect (carbonate 3 "Fizz")
        >> connect (carbonate 5 "Buzz")
        >> either (printf "%s; ") (printf "%i; ")

    // test
    [1..100] |> List.iter fizzBuzz
```

Carbonation as failure?

We defined carbonation as "success"in the example above -- it seems the natural thing to do, surely. But if you recall, in the railway oriented programming model, "success" means that data should be passed on to the next function, while "failure" means to bypass all the intermediate functions and go straight to the end.

For FizzBuzz, the "bypass all the intermediate functions" track is the track with the carbonated labels, while the "pass on to next function" track is the one with integers.

So we should really reverse the tracks: "Failure" now means carbonation, while "Success" means no carbonation.

By doing this, we also get to reuse the pre-defined `bind` function, rather than having to write our own `connect` function.

Here's the code with the tracks switched around:

```
module FizzBuzz_RailwayOriented_CarbonationIsFailure =

    open RailwayCombinatorModule

    // carbonate a value
    let carbonate factor label i =
        if i % factor = 0 then
            fail label
        else
            succeed i

    let fizzBuzz =
        carbonate 15 "FizzBuzz"
        >> bind (carbonate 3 "Fizz")
        >> bind (carbonate 5 "Buzz")
        >> either (printf "%i; ") (printf "%s; ")

    // test
    [1..100] |> List.iter fizzBuzz
```

What are the two tracks anyway?

The fact that we can swap the tracks so easily implies that that maybe there is a weakness in the design. Are we trying to use a design that doesn't fit?

Why does one track have to be "Success" and another track "Failure" anyway? It doesn't seem to make much difference.

So, why don't we *keep* the two-track idea, but get rid of the "Success" and "Failure" labels.

Instead, we can call one track "Carbonated" and the other "Uncarbonated".

To make this work, we can define an active pattern and constructor methods, just as we did for "Success/Failure".

```
let (|Uncarbonated|Carbonated|) =
    function
    | Choice1Of2 u -> Uncarbonated u
    | Choice2Of2 c -> Carbonated c

/// convert a single value into a two-track result
let uncarbonated x = Choice1Of2 x
let carbonated x = Choice2Of2 x
```

If you are doing domain driven design, it is good practice to write code that uses the appropriate Ubiquitous Language rather than language that is not applicable.

In this case, if FizzBuzz was our domain, then our functions could now use the domain-friendly terminology of `carbonated` and `uncarbonated` rather than "success" or "failure".

```
let carbonate factor label i =
    if i % factor = 0 then
        carbonated label
    else
        uncarbonated i

let connect f =
    function
    | Uncarbonated i -> f i
    | Carbonated x -> carbonated x
```

Note that, as before, the `connect` function can be rewritten using `either` (or we can just use the predefined `bind` as before):

```
let connect' f =
    either f carbonated
```

Here's all the code in one module:

```
module FizzBuzz_RailwayOriented_UsingCustomChoice =

    open RailwayCombinatorModule

    let (|Uncarbonated|Carbonated|) =
        function
        | Choice1Of2 u -> Uncarbonated u
        | Choice2Of2 c -> Carbonated c

    /// convert a single value into a two-track result
    let uncarbonated x = Choice1Of2 x
    let carbonated x = Choice2Of2 x

    // carbonate a value
    let carbonate factor label i =
        if i % factor = 0 then
            carbonated label
        else
            uncarbonated i

    let connect f =
        function
        | Uncarbonated i -> f i
        | Carbonated x -> carbonated x

    let connect' f =
        either f carbonated

    let fizzBuzz =
        carbonate 15 "FizzBuzz"
        >> connect (carbonate 3 "Fizz")
        >> connect (carbonate 5 "Buzz")
        >> either (printf "%i; ") (printf "%s; ")

    // test
    [1..100] |> List.iter fizzBuzz
```

Adding rules

There are some problems with the version we have so far:

- The "15" test is ugly. Can we get rid of it and reuse the "3" and "5" cases?
- The "3" and "5" cases are hard-coded. Can we make this more dynamic?

As it happens, we can kill two birds with one stone and address both of these issues at once.

Instead of combining all the "switch" functions in *series*, we can "add" them together in *parallel*. In the railway oriented programming post, we used this technique for combining validation functions. For FizzBuzz we are going to use it for doing all the factors at once.

The trick is to define a "append" or "concat" function for combining two functions. Once we can add two functions this way, we can continue and add as many as we like.

So given that we have two carbonation functions, how do we concat them?

Well, here are the possible cases:

- If they both have carbonated outputs, we concatenate the labels into a new carbonated label.
- If one has a carbonated output and the other doesn't, then we use the carbonated one.
- If neither has a carbonated output, then we use either uncarbonated one (they will be the same).

Here's the code:

```
// concat two carbonation functions
let (<+>) switch1 switch2 x =
    match (switch1 x),(switch2 x) with
    | Carbonated s1,Carbonated s2 -> carbonated (s1 + s2)
    | Uncarbonated f1,Carbonated s2  -> carbonated s2
    | Carbonated s1,Uncarbonated f2 -> carbonated s1
    | Uncarbonated f1,Uncarbonated f2 -> uncarbonated f1
```

As an aside, notice that this code is almost like math, with `uncarbonated` playing the role of "zero", like this:

```
something + something = combined somethings
zero + something = something
something + zero = something
zero + zero = zero
```

This is not a coincidence! We will see this kind of thing pop up over and over in functional code. I'll be talking about this in a future post.

Anyway, with this "concat" function in place, we can rewrite the main `fizzBuzz` like this:

```
let fizzBuzz =
    let carbonateAll =
        carbonate 3 "Fizz" <+> carbonate 5 "Buzz"

    carbonateAll
    >> either (printf "%i; ") (printf "%s; ")
```

The two `carbonate` functions are added and then passed to `either` as before.

Here is the complete code:

```
module FizzBuzz_RailwayOriented_UsingAppend =

    open RailwayCombinatorModule

    let (|Uncarbonated|Carbonated|) =
        function
        | Choice1Of2 u -> Uncarbonated u
        | Choice2Of2 c -> Carbonated c

    /// convert a single value into a two-track result
    let uncarbonated x = Choice1Of2 x
    let carbonated x = Choice2Of2 x

    // concat two carbonation functions
    let (<+>) switch1 switch2 x =
        match (switch1 x),(switch2 x) with
        | Carbonated s1,Carbonated s2 -> carbonated (s1 + s2)
        | Uncarbonated f1,Carbonated s2  -> carbonated s2
        | Carbonated s1,Uncarbonated f2 -> carbonated s1
        | Uncarbonated f1,Uncarbonated f2 -> uncarbonated f1

    // carbonate a value
    let carbonate factor label i =
        if i % factor = 0 then
            carbonated label
        else
            uncarbonated i

    let fizzBuzz =
        let carbonateAll =
            carbonate 3 "Fizz" <+> carbonate 5 "Buzz"

        carbonateAll
        >> either (printf "%i; ") (printf "%s; ")

    // test
    [1..100] |> List.iter fizzBuzz
```

With this addition logic available, we can easily refactor the code to use rules. Just as with the earlier "pipeline" implementation, we can use `reduce` to add all the rules together at once.

Here's the version with rules:

```
module FizzBuzz_RailwayOriented_UsingAddition =

    // code as above

    let fizzBuzzPrimes rules =
        let carbonateAll  =
            rules
            |> List.map (fun (factor,label) -> carbonate factor label)
            |> List.reduce (<+>)

        carbonateAll
        >> either (printf "%i; ") (printf "%s; ")

    // test
    let rules = [ (3,"Fizz"); (5,"Buzz"); (7,"Baz") ]
    [1..105] |> List.iter (fizzBuzzPrimes rules)
```

Summary

In this post, we've seen three different implementations:

- An imperative version that used mutable values and mixed side-effects with logic.
- A "pipeline" version that passed a data structure through a series of functions.
- A "railway oriented" version that had two separate tracks, and used "addition" to combine functions in parallel.

In my opinion, the imperative version is the worst design. Even though it was easy to hack out quickly, it has a number of problems that make it brittle and error-prone.

Of the two functional versions, I think the "railway oriented" version is cleaner, for this problem at least.

By using the `Choice` type rather than a tuple or special record, we made the code more elegant thoughout. You can see the difference if you compare the pipeline version of `carbonate` with the railway oriented version of `carbonate`.

In other situations, of course, the railway oriented approach might not work, and the pipeline approach might be better. I hope this post has given some useful insight into both.

If you're a FizzBuzz fan, check out the Functional Reactive Programming page, which has yet another variant of the problem.

Postscript: Be careful when using List.reduce

Be careful with `List.reduce` -- it will fail with empty lists. So if you have an empty rule set, the code will throw a `System.ArgumentException`.

In the pipeline case, you can see this by adding the following snippet to the module:

```
module FizzBuzz_Pipeline_WithRules =

    // code as before

    // bug
    let emptyRules = []
    [1..105] |> List.iter (fizzBuzz emptyRules)
```

The fix is to replace `List.reduce` with `List.fold`. `List.fold` requires an additional parameter: the initial (or "zero") value. In this case, we can use the identity function, `id`, as the initial value.

Here is the fixed version of the code:

```
let allRules =
    rules
    |> List.map (fun (factor,label) -> carbonate factor label)
    |> List.fold (>>) id
```

Similarly, in the railway oriented example, we had:

```
let allRules =
    rules
    |> List.map (fun (factor,label) -> carbonate factor label)
    |> List.reduce (<+>)
```

which should be corrected to:

```
let allRules =
    rules
    |> List.map (fun (factor,label) -> carbonate factor label)
    |> List.fold (<+>) zero
```

where `zero` is the "default" function to use if the list is empty.

As an exercise, define the zero function for this case. (Hint: we have actually already defined it under another name).

In this series, we'll look at a very common "pattern" known as a *monoid*.

Monoids are not really a design pattern; more an approach to working with many different types of values in a common way. In fact, once you understand monoids, you will start seeing them everywhere!

- Monoids without tears. A mostly mathless discussion of a common functional pattern.
- Monoids in practice. Monoids without tears - Part 2.
- Working with non-monoids. Monoids without tears - Part 3.

Monoids without tears

If you are coming from an OO background, one of the more challenging aspects of learning functional programming is the lack of obvious design patterns. There are plenty of idioms such as partial application, and error handling techniques, but no apparent patterns in the GoF sense.

In this post, we'll look at a very common "pattern" known as a *monoid*. Monoids are not really a design pattern; more an approach to working with many different types of values in a common way. In fact, once you understand monoids, you will start seeing them everywhere!

Unfortunately the term "monoid" itself is a bit off-putting. It originally comes from mathematics but the concept as applied to programming is easy to grasp without any math at all, as I hope to demonstrate. In fact, if we were to name the concept today in a programming context, we might call it something like `ICombinable` instead, which is not nearly as scary.

Finally, you might be wondering if a "monoid" has any connection with a "monad". Yes, there is a mathematical connection between them, but in programming terms, they are very different things, despite having similar names.

Uh-oh... some equations

On this site, I generally don't use any math, but in this case I'm going to break my self-imposed rule and show you some equations.

Ready? Here's the first one:

```
1 + 2 = 3
```

Could you handle that? How about another one?

```
1 + (2 + 3) = (1 + 2) + 3
```

And finally one more...

```
1 + 0 = 1 and 0 + 1 = 1
```

Ok! We're done! If you can understand these equations, then you have all the math you need to understand monoids.

Thinking like a mathematician

> *"A mathematician, like a painter or poet, is a maker of patterns. If his patterns are more permanent than theirs, it is because they are made with ideas" -- G H Hardy*

Most people imagine that mathematicians work with numbers, doing complicated arithmetic and calculus.

This is a misconception. For example, if you look at typical high-level math discussions, you will see lots of strange words, and lots of letter and symbols, but not a lot of arithmetic.

One of the things that mathematicians *do* do though, is try to find patterns in things. "What do these things have in common?" and "How can we generalize these concepts?" are typical mathematical questions.

So let's look at these three equations through a mathematician's eyes.

Generalizing the first equation

A mathematician would look at `1 + 2 = 3` and think something like:

- We've got a bunch of things (integers in this case)
- We've got some way of combining two of them (addition in this case)
- And the result is another one of these things (that is, we get another integer)

And then a mathematician might try to see if this pattern could be generalized to other kinds of things and operations.

Let's start by staying with integers as the "things". What other ways are there of combining integers? And do they fit the pattern?

Let's try multiplication, does that fit this pattern?

The answer is yes, multiplication does fit this pattern because multiplying any two integers results in another integer.

What about division? Does that fit the pattern? The answer is no, because in most cases, dividing two integers results in a fraction, which is *not* an integer (I'm ignoring integer division).

What about the `max` function? Does that fit the pattern? It combines two integers and returns one of them, so the answer is yes.

What about the `equals` function? It combines two integers but returns a boolean, not an integer, so the answer is no.

Enough of integers! What other kinds of things can we think of?

Floats are similar to integers, but unlike integers, using division with floats does result in another float, so the division operation fits the pattern.

How about booleans? They can be combined using operators such as AND, OR and so on. Does `aBool AND aBool` result in another bool? Yes! And `OR` too fits the pattern.

Strings next. How can they be combined? One way is string concatenation, which returns another string, which is what we want. But something like the equality operation doesn't fit, because it returns a boolean.

Finally, let's consider lists. As for strings, the obvious way to combine them is with list concatenation, which returns another list and fits the pattern.

We can continue on like this for all sorts of objects and combining operations, but you should see how it works now.

You might ask: why is it so important that the operation return another thing of the same type? The answer is that **you can chain together multiple objects using the operation**.

For example, because `1 + 2` is another integer, you can add 3 to it. And then because `1 + 2 + 3` is an integer as well, you can keep going and add say, 4, to the result. In other words, it is only because integer addition fits the pattern that you can write a sequence of additions like this: `1 + 2 + 3 + 4`. You couldn't write `1 = 2 = 3 = 4` in the same way, because integer equality doesn't fit the pattern.

And of course, the chain of combined items can be as long as we like. In other words, this pattern allows us to extend a pairwise operation into **an operation that works on lists**.

Mathematicians call the requirement that "the result is another one of these things" the *closure* requirement.

Generalizing the second equation

Ok, what about the next equation, `1 + (2 + 3) = (1 + 2) + 3`? Why is that important?

Well, if you think about the first pattern, it says we can build up a chain of operations such as `1 + 2 + 3`. But we have only got a pairwise operation. So what order should we do the combining in? Should we combine 1 and 2 first, then combine the result with 3? Or should we combine 2 and 3 first and then combine 1 with that result? Does it make a difference?

That's where this second equation is useful. It says that, for addition, the order of combination doesn't matter. You get the same result either way.

So for a chain of four items like this: `1 + 2 + 3 + 4`, we could start working from the left hand side: `((1+2) + 3) + 4` or from the right hand side: `1 + (2 + (3+4))` or even do it in two parts and then combine them like this: `(1+2) + (3+4)`.

Let's see if this pattern applies to the examples we've already looked at.

Again, let's start with other ways of combining integers.

We'll start with multiplication again. Does `1 * (2 * 3)` give the same result as `(1 * 2) * 3`? Yes. Just as with addition, the order doesn't matter.

Now let's try subtraction. Does `1 - (2 - 3)` give the same result as `(1 - 2) - 3`? No. For subtraction, the order *does* matter.

What about division? Does `12 / (2 / 3)` give the same result as `(12 / 2) / 3`? No. For division also, the order matters.

But the `max` function does work. `max( max(12,2), 3)` gives the same result as `max(12, max(2,3)`.

What about strings and lists? Does concatenation meet the requirement? What do you think?

Here's a question... Can we come up with an operation for strings that *is* order dependent?

Well, how about a function like "subtractChars" which removes all characters in the right string from the left string. So `subtractChars("abc","ab")` is just `"c"`. `subtractChars` is indeed order dependent, as you can see with a simple example: `subtractChars("abc", subtractChars("abc","abc"))` is not the same string as `subtractChars(subtractChars("abc","abc"),"abc")`.

Mathematicians call the requirement that "the order doesn't matter" the *associativity* requirement.

Important Note: When I say the "order of combining", I am talking about the order in which you do the pairwise combining steps -- combining one pair, and then combining the result with the next item.

But it is critical that the overall sequence of the items be left unchanged. This is because for certain operations, if you change the sequencing of the items, then you get a completely different result! `1 - 2` does not mean the same as `2 - 1` and `2 / 3` does not mean the same as `3 / 2`.

Of course, in many common cases, the sequence order doesn't matter. After all, `1+2` is the same as `2+1`. In this case, the operation is said to be *commutative*.

The third equation

Now let's look at the third equation, `1 + 0 = 1`.

A mathematician would say something like: that's interesting -- there is a special kind of thing ("zero") that, when you combine it with something, just gives you back the original something, as if nothing had happened.

So once more, let's revisit our examples and see if we can extend this "zero" concept to other operations and other things.

Again, let's start with multiplication. Is there some value, such that when you multiply a number with it, you get back the original number?

Yes, of course! The number one. So for multiplication, the number `1` is the "zero".

What about `max`? Is there a "zero" for that? For 32 bit ints, yes. Combining `System.Int32.MinValue` with any other 32 bit integer using `max` will return the other integer. That fits the definition of "zero" perfectly.

What about booleans combined using AND? Is there a zero for that? Yes. It is the value `True`. Why? Because `True AND False` is `False`, and `True AND True` is `True`. In both cases the other value is returned untouched.

What about booleans combined using OR? Is there a zero for that as well? I'll let you decide.

Moving on, what about string concatenation? Is there a "zero" for this? Yes, indeed -- it is just the empty string.

```
"" + "hello" = "hello"
"hello" + "" = "hello"
```

Finally, for list concatenation, the "zero" is just the empty list.

```
[] @ [1;2;3] = [1;2;3]
[1;2;3] @ [] = [1;2;3]
```

You can see that the "zero" value depends very much on the operation, not just on the set of things. The zero for integer addition is different from the "zero" for integer multiplication, which is different again from the from "zero" for `Max`.

Mathematicians call the "zero" the *identity element*.

The equations revisited

So now let's revisit the equations with our new generalizations in mind.

Before, we had:

```
1 + 2 = 3
1 + (2 + 3) = (1 + 2) + 3
1 + 0 = 1 and 0 + 1 = 1
```

But now we have something much more abstract, a set of generalized requirements that can apply to all sorts of things:

- You start with a bunch of things, *and* some way of combining them two at a time.
- **Rule 1 (Closure)**: The result of combining two things is always another one of the things.
- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.
- **Rule 3 (Identity element)**: There is a special thing called "zero" such that when you combine any thing with "zero" you get the original thing back.

With these rules in place, we can come back to the definition of a monoid. A "monoid" is just a system that obeys all three rules. Simple!

As I said at the beginning, don't let the mathematical background put you off. If programmers had named this pattern, it probably would been called something like "the combinable pattern" rather than "monoid". But that's life. The terminology is already well-established, so we have to use it.

Note there are *two* parts to the definition of a monoid -- the things plus the associated operation. A monoid is not just "a bunch of things", but "a bunch of things" *and* "some way of combining them". So, for example, "the integers" is not a monoid, but "the integers under addition" is a monoid.

Semigroups

In certain cases, you have a system that only follows the first two rules, and there is no candidate for a "zero" value.

For example, if your domain consists only of strictly positive numbers, then under addition they are closed and associative, but there is no positive number that can be "zero".

Another example might be the intersection of finite lists. It is closed and associative, but there is no (finite) list that when intersected with any other finite list, leaves it untouched.

This kind of system still quite useful, and is called a "semigroup" by mathematicians, rather than a monoid. Luckily, there is a trick that can convert any semigroup into a monoid (which I'll describe later).

A table of classifications

Let's put all our examples into a table, so you can see them all together.

Things	Operation	Closed?	Associative?	Identity?	Classifi
Int32	Addition	Yes	Yes	0	Monoid
Int32	Multiplication	Yes	Yes	1	Monoid
Int32	Subtraction	Yes	No	0	Other
Int32	Max	Yes	Yes	Int32.MinValue	Monoid
Int32	Equality	No			Other
Int32	Less than	No			Other
Float	Multiplication	Yes	No (See note 1)	1	Other
Float	Division	Yes (See note 2)	No	1	Other
Positive Numbers	Addition	Yes	Yes	No identity	Semigro
Positive Numbers	Multiplication	Yes	Yes	1	Monoid
Boolean	AND	Yes	Yes	true	Monoid
Boolean	OR	Yes	Yes	false	Monoid
String	Concatenation	Yes	Yes	Empty string ""	Monoid
String	Equality	No			Other
String	"subtractChars"	Yes	No	Empty string ""	Other
List	Concatenation	Yes	Yes	Empty list []	Monoid
List	Intersection	Yes	Yes	No identity	Semigro

There are many other kinds of things you can add to this list; polynomials, matrices, probability distributions, and so on. This post won't discuss them, but once you get the idea of monoids, you will see that the concept can be applied to all sorts of things.

[Note 1] As Doug points out in the comments, floats are not associative. Replace 'float' with 'real number' to get associativity.

[Note 2] Mathematical real numbers are not closed under division, because you cannot divide by zero and get another real number. However, with IEEE floating point numbers you *can* divide by zero and get a valid value. So floats are indeed closed under division! Here's a demonstration:

```
let x = 1.0/0.0 // infinity
let y = x * 2.0 // two times infinity
let z = 2.0 / x // two divided by infinity
```

What use are monoids to a programmer?

So far, we have described some abstract concepts, but what good are they for real-world programming problems?

The benefit of closure

As we've seen, the closure rule has the benefit that you can convert pairwise operations into operations that work on lists or sequences.

In other words, if we can define a pairwise operation, we can extend it to list operations "for free".

The function that does this is typically called "reduce". Here are some examples:

Explicit	Using reduce
`1 + 2 + 3 + 4`	`[ 1; 2; 3; 4 ] |> List.reduce (+)`
`1 * 2 * 3 * 4`	`[ 1; 2; 3; 4 ] |> List.reduce (*)`
`"a" + "b" + "c" + "d"`	`[ "a"; "b"; "c"; "d" ] |> List.reduce (+)`
`[1] @ [2] @ [3] @ [4]`	`[ [1]; [2]; [3]; [4] ] |> List.reduce (@)`

You can see that `reduce` can be thought of as inserting the specified operation between each element of the list.

Note that in the last example, the input to `reduce` is a list of lists, and the output is a single list. Make sure you understand why this is.

The benefit of associativity

If the pairwise combinations can be done in any order, that opens up some interesting implementation techniques, such as:

- Divide and conquer algorithms
- Parallelization
- Incrementalism

These are deep topics, but let's have a quick look!

Divide and conquer algorithms

Consider the task of summing the first 8 integers; how could we implement this?

One way would be a crude step-by-step sum, as follows:

```
let sumUpTo2 = 1 + 2
let sumUpTo3 = sumUpTo2 + 3
let sumUpTo4 = sumUpTo3 + 4
// etc
let result = sumUpTo7 + 8
```

But because the sums can be done in any order, we could also implement the requirement by splitting the sum into two halves, like this

```
let sum1To4 = 1 + 2 + 3 + 4
let sum5To8 = 5 + 6 + 7 + 8
let result = sum1To4 + sum5To8
```

and then we can recursively split the sums into sub-sums in the same way until we get down to the basic pairwise operation:

```
let sum1To2 = 1 + 2
let sum3To4 = 3 + 4
let sum1To4 = sum1To2 + sum3To4
```

This "divide and conquer" approach may seem like overkill for something like a simple sum, but we'll see in a future post that, in conjunction with a `map`, it is the basis for some well known aggregation algorithms.

Parallelization

Once we have a divide and conquer strategy, it can be easily converted into a parallel algorithm.

For example, to sum the first 8 integers on a four-core CPU, we might do something like this:

	Core 1	Core 2	Core 3	Core 4
Step 1	`sum12 = 1 + 2`	`sum34 = 3 + 4`	`sum56 = 5 + 6`	`sum78 = 7 + 8`
Step 2	`sum1234 = sum12 + sum34`	`sum5678 = sum56 + sum78`	(idle)	(idle)
Step 3	`sum1234 + sum5678`	(idle)	(idle)	(idle)

There are still seven calculations that need to be done, but because we are doing it parallel, we can do them all in three steps.

Again, this might seem like a trivial example, but big data systems such as Hadoop are all about aggregating large amounts of data, and if the aggregation operation is a monoid, then you can, in theory, easily scale these aggregations by using multiple machines*.

*In practice, of course, the devil is in the details, and real-world systems don't work exactly this way.

Incrementalism

Even if you do not need parallelism, a nice property of monoids is that they support incremental calculations.

For example, let's say you have asked me to calculate the sum of one to five. Then of course I give you back the answer fifteen.

But now you say that you have changed your mind, and you want the sum of one to *six* instead. Do I have to add up all the numbers again, starting from scratch? No, I can use the previous sum, and just add six to it incrementally. This is possible because integer addition is a monoid.

That is, when faced with a sum like `1 + 2 + 3 + 4 + 5 + 6`, I can group the numbers any way I like. In particular, I can make an incremental sum like this: `(1 + 2 + 3 + 4 + 5) + 6`, which then reduces to `15 + 6`.

In this case, recalculating the entire sum from scratch might not be a big deal, but consider a real-world example like web analytics, counting the number of visitors over the last 30 days, say. A naive implementation might be to calculate the numbers by parsing the logs of the last 30 days data. A more efficient approach would be to recognize that the previous 29 days have not changed, and to only process the incremental changes for one day. As a result, the parsing effort is greatly reduced.

Similarly, if you had a word count of a 100 page book, and you added another page, you shouldn't need to parse all 101 pages again. You just need to count the words on the last page and add that to the previous total.*

- Technically, these are scary sounding *monoid homomorphisms*. I will explain what this is in the next post.

The benefit of identity

Having an identity element is not always required. Having a closed, associative operation (i.e. a semigroup) is sufficient to do many useful things.

But in some cases, it is not enough. For example, here are some cases that might crop up:

- How can I use `reduce` on an empty list?
- If I am designing a divide and conquer algorithm, what should I do if one of the "divide" steps has nothing in it?
- When using an incremental algorithm, what value should I start with when I have no data?

In all cases we need a "zero" value. This allows us to say, for example, that the sum of an empty list is `0`.

Regarding the first point above, if we are concerned that the list might be empty, then we must replace `reduce` with `fold`, which allows an initial value to be passed in. (Of course, `fold` can be used for more things than just monoid operations.)

Here are `reduce` and `fold` in action:

```
// ok
[1..10] |> List.reduce (+)

// error
[] |> List.reduce (+)

// ok with explicit zero
[1..10] |> List.fold (+) 0

// ok with explicit zero
[] |> List.fold (+) 0
```

Using a "zero" can result in counter-intuitive results sometimes. For example, what is the *product* of an empty list of integers?

The answer is `1`, not `0` as you might expect! Here's the code to prove it:

```
[1..4] |> List.fold (*) 1  // result is 24
[] |> List.fold (*) 1      // result is 1
```

Summary of the benefits

To sum up, a monoid is basically a way to describe an aggregation pattern -- we have a list of things, we have some way of combining them, and we get a single aggregated object back at the end.

Or in F# terms:

```
Monoid Aggregation : 'T list -> 'T
```

So when you are designing code, and you start using terms like "sum", "product", "composition", or "concatenation", these are clues that you are dealing with a monoid.

Next steps

Now that we understand what a monoid is, let's see how they can be used in practice.

In the next post in this series, we'll look at how you might write real code that implements the monoid "pattern".

Monoids in practice

In the previous post, we looked at the definition of a monoid. In this post, we'll see how to implement some monoids.

First, let's revisit the definition:

- You start with a bunch of things, *and* some way of combining them two at a time.
- **Rule 1 (Closure)**: The result of combining two things is always another one of the things.
- **Rule 2 (Associativity)**: When combining more than two things, which pairwise combination you do first doesn't matter.
- **Rule 3 (Identity element)**: There is a special thing called "zero" such that when you combine any thing with "zero" you get the original thing back.

For example, if strings are the things, and string concatenation is the operation, then we have a monoid. Here's some code that demonstrates this:

```
let s1 = "hello"
let s2 = " world!"

// closure
let sum = s1 + s2  // sum is a string

// associativity
let s3 = "x"
let s4a = (s1+s2) + s3
let s4b = s1 + (s2+s3)
assert (s4a = s4b)

// an empty string is the identity
assert (s1 + "" = s1)
assert ("" + s1 = s1)
```

But now let's try to apply this to a more complicated object.

Say that we have have an `OrderLine`, a little structure that represents a line in a sales order, say.

```
type OrderLine = {
    ProductCode: string
    Qty: int
    Total: float
    }
```

And then perhaps we might want to find the total for an order, that is, we want to sum the `Total` field for a list of lines.

The standard imperative approach would be to create a local `total` variable, and then loop through the lines, summing as we go, like this:

```
let calculateOrderTotal lines =
    let mutable total = 0.0
    for line in lines do
        total <- total + line.Total
    total
```

Let's try it:

```
module OrdersUsingImperativeLoop =

    type OrderLine = {
        ProductCode: string
        Qty: int
        Total: float
        }

    let calculateOrderTotal lines =
        let mutable total = 0.0
        for line in lines do
            total <- total + line.Total
        total

    let orderLines = [
        {ProductCode="AAA"; Qty=2; Total=19.98}
        {ProductCode="BBB"; Qty=1; Total=1.99}
        {ProductCode="CCC"; Qty=3; Total=3.99}
        ]

    orderLines
    |> calculateOrderTotal
    |> printfn "Total is %g"
```

But of course, being an experienced functional programmer, you would sneer at this, and use `fold` in `calculateOrderTotal` instead, like this:

```
module OrdersUsingFold =

    type OrderLine = {
        ProductCode: string
        Qty: int
        Total: float
        }

    let calculateOrderTotal lines =
        let accumulateTotal total line =
            total + line.Total
        lines
        |> List.fold accumulateTotal 0.0

    let orderLines = [
        {ProductCode="AAA"; Qty=2; Total=19.98}
        {ProductCode="BBB"; Qty=1; Total=1.99}
        {ProductCode="CCC"; Qty=3; Total=3.99}
        ]

    orderLines
    |> calculateOrderTotal
    |> printfn "Total is %g"
```

So far, so good. Now let's look at a solution using a monoid approach.

For a monoid, we need to define some sort of addition or combination operation. How about something like this?

```
let addLine orderLine1 orderLine2 =
    orderLine1.Total + orderLine2.Total
```

But this is no good, because we forgot a key aspect of monoids. The addition must return a value of the same type!

If we look at the signature for the `addLine` function...

```
addLine : OrderLine -> OrderLine -> float
```

...we can see that the return type is `float` not `OrderLine`.

What we need to do is return a whole other `OrderLine`. Here's a correct implementation:

```
let addLine orderLine1 orderLine2 =
    {
    ProductCode = "TOTAL"
    Qty = orderLine1.Qty + orderLine2.Qty
    Total = orderLine1.Total + orderLine2.Total
    }
```

Now the signature is correct: `addLine : OrderLine -> OrderLine -> OrderLine`.

Note that because we have to return the entire structure we have to specify something for the `ProductCode` and `Qty` as well, not just the total. The `Qty` is easy, we can just do a sum. For the `ProductCode`, I decided to use the string "TOTAL", because we don't have a real product code we can use.

Let's give this a little test:

```
// utility method to print an OrderLine
let printLine {ProductCode=p; Qty=q;Total=t} =
    printfn "%-10s %5i %6g" p q t

let orderLine1 = {ProductCode="AAA"; Qty=2; Total=19.98}
let orderLine2 = {ProductCode="BBB"; Qty=1; Total=1.99}

//add two lines to make a third
let orderLine3 = addLine orderLine1 orderLine2
orderLine3 |> printLine // and print it
```

We should get this result:

```
TOTAL          3  21.97
```

NOTE: For more on the printf formatting options used, see the post on printf here.

Now let's apply this to a list using `reduce`:

```
let orderLines = [
    {ProductCode="AAA"; Qty=2; Total=19.98}
    {ProductCode="BBB"; Qty=1; Total=1.99}
    {ProductCode="CCC"; Qty=3; Total=3.99}
    ]

orderLines
|> List.reduce addLine
|> printLine
```

With the result:

```
TOTAL           6  25.96
```

At first, this might seem like extra work, and just to add up a total. But note that we now have more information than just the total; we also have the sum of the qtys as well.

For example, we can easily reuse the `printLine` function to make a simple receipt printing function that includes the total, like this:

```
let printReceipt lines =
    lines
    |> List.iter printLine

    printfn "-----------------------"

    lines
    |> List.reduce addLine
    |> printLine

orderLines
|> printReceipt
```

Which gives an output like this:

```
AAA             2  19.98
BBB             1   1.99
CCC             3   3.99
-----------------------
TOTAL           6  25.96
```

More importantly, we can now use the incremental nature of monoids to keep a running subtotal that we update every time a new line is added.

Here's an example:

```
let subtotal = orderLines |> List.reduce addLine
let newLine = {ProductCode="DDD"; Qty=1; Total=29.98}
let newSubtotal = subtotal |> addLine newLine
newSubtotal |> printLine
```

We could even define a custom operator such as `++` so that we can add lines together naturally as it they were numbers:

```
let (++) a b = addLine a b  // custom operator

let newSubtotal = subtotal ++ newLine
```

You can see that using the monoid pattern opens up a whole new way of thinking. You can apply this "add" approach to almost any kind of object.

For example, what would a product "plus" a product look like? Or a customer "plus" a customer? Let your imagination run wild!

Are we there yet?

You might have noticed that we not quite done yet. There is a third requirement for a monoid that we haven't discussed yet -- the zero or identity element.

In this case, the requirement means that we need some kind of `OrderLine` such that adding it to another order line would leave the original untouched. Do we have such a thing?

Right now, no, because the addition operation always changes the product code to "TOTAL". What we have right now is in fact a *semigroup*, not a monoid.

As you can see, a semigroup is perfectly useable. But a problem would arise if we had an empty list of lines and we wanted to total them. What should the result be?

One workaround would be to change the `addLine` function to ignore empty product codes. And then we could use an order line with an empty code as the zero element.

Here's what I mean:

```
let addLine orderLine1 orderLine2 =
    match orderLine1.ProductCode, orderLine2.ProductCode with
    // is one of them zero? If so, return the other one
    | "", _ -> orderLine2
    | _, "" -> orderLine1
    // anything else is as before
    | _ ->
        {
        ProductCode = "TOTAL"
        Qty = orderLine1.Qty + orderLine2.Qty
        Total = orderLine1.Total + orderLine2.Total
        }

let zero = {ProductCode=""; Qty=0; Total=0.0}
let orderLine1 = {ProductCode="AAA"; Qty=2; Total=19.98}
```

We can then test that identity works as expected:

```
assert (orderLine1 = addLine orderLine1 zero)
assert (orderLine1 = addLine zero orderLine1)
```

This does seem a bit hacky, so I wouldn't recommend this technique in general. There's another way to get an identity that we'll be discussing later.

Introducing a special total type

In the example above, the `OrderLine` type was very simple and it was easy to overload the fields for the total.

But what would happen if the `OrderLine` type was more complicated? For example, if it had a `Price` field as well, like this:

```
type OrderLine = {
    ProductCode: string
    Qty: int
    Price: float
    Total: float
    }
```

Now we have introduced a complication. What should we set the `Price` to when we combine two lines? The average price? No price?

```
let addLine orderLine1 orderLine2 =
    {
    ProductCode = "TOTAL"
    Qty = orderLine1.Qty + orderLine2.Qty
    Price = 0 // or use average price?
    Total = orderLine1.Total + orderLine2.Total
    }
```

Neither seems very satisfactory.

The fact that we don't know what to do probably means that our design is wrong.

Really, we only need a subset of the data for the total, not all of it. How can we represent this?

With a discriminated union of course! One case can be used for product lines, and the other case can be used for totals only.

Here's what I mean:

```
type ProductLine = {
    ProductCode: string
    Qty: int
    Price: float
    LineTotal: float
    }

type TotalLine = {
    Qty: int
    OrderTotal: float
    }

type OrderLine =
    | Product of ProductLine
    | Total of TotalLine
```

This design is much nicer. We now have a special structure just for totals and we don't have to use contortions to make the excess data fit. We can even remove the dummy "TOTAL" product code.

Note that I named the "total" field differently in each record. Having unique field names like this means that you don't have to always specify the type explicitly.

Unfortunately, the addition logic is more complicated now, as we have to handle every combination of cases:

```
let addLine orderLine1 orderLine2 =
    let totalLine =
        match orderLine1,orderLine2 with
        | Product p1, Product p2 ->
            {Qty = p1.Qty + p2.Qty;
            OrderTotal = p1.LineTotal + p2.LineTotal}
        | Product p, Total t ->
            {Qty = p.Qty + t.Qty;
            OrderTotal = p.LineTotal + t.OrderTotal}
        | Total t, Product p ->
            {Qty = p.Qty + t.Qty;
            OrderTotal = p.LineTotal + t.OrderTotal}
        | Total t1, Total t2 ->
            {Qty = t1.Qty + t2.Qty;
            OrderTotal = t1.OrderTotal + t2.OrderTotal}
    Total totalLine // wrap totalLine to make OrderLine
```

Note that we cannot just return the `TotalLine` value. We have to wrap in the `Total` case to make a proper `OrderLine`. If we didn't do that, then our `addLine` would have the signature `OrderLine -> OrderLine -> TotalLine`, which is not correct. We have to have the signature `OrderLine -> OrderLine -> OrderLine` -- nothing else will do!

Now that we have two cases, we need to handle both of them in the `printLine` function:

```
let printLine =  function
    | Product {ProductCode=p; Qty=q; Price=pr; LineTotal=t} ->
        printfn "%-10s %5i @%4g each %6g" p q pr t
    | Total {Qty=q; OrderTotal=t} ->
        printfn "%-10s %5i            %6g" "TOTAL" q t
```

But once we have done this, we can now use addition just as before:

```
let orderLine1 = Product {ProductCode="AAA"; Qty=2; Price=9.99; LineTotal=19.98}
let orderLine2 = Product {ProductCode="BBB"; Qty=1; Price=1.99; LineTotal=1.99}
let orderLine3 = addLine orderLine1 orderLine2

orderLine1 |> printLine
orderLine2 |> printLine
orderLine3 |> printLine
```

Identity again

Again, we haven't dealt with the identity requirement. We could try using the same trick as before, with a blank product code, but that only works with the `Product` case.

To get a proper identity, we really need to introduce a *third* case, `EmptyOrder` say, to the union type:

```
type ProductLine = {
    ProductCode: string
    Qty: int
    Price: float
    LineTotal: float
    }

type TotalLine = {
    Qty: int
    OrderTotal: float
    }

type OrderLine =
    | Product of ProductLine
    | Total of TotalLine
    | EmptyOrder
```

With this extra case available, we rewrite the `addLine` function to handle it:

```
let addLine orderLine1 orderLine2 =
    match orderLine1,orderLine2 with
    // is one of them zero? If so, return the other one
    | EmptyOrder, _ -> orderLine2
    | _, EmptyOrder -> orderLine1
    // otherwise as before
    | Product p1, Product p2 ->
        Total { Qty = p1.Qty + p2.Qty;
        OrderTotal = p1.LineTotal + p2.LineTotal}
    | Product p, Total t ->
        Total {Qty = p.Qty + t.Qty;
        OrderTotal = p.LineTotal + t.OrderTotal}
    | Total t, Product p ->
        Total {Qty = p.Qty + t.Qty;
        OrderTotal = p.LineTotal + t.OrderTotal}
    | Total t1, Total t2 ->
        Total {Qty = t1.Qty + t2.Qty;
        OrderTotal = t1.OrderTotal + t2.OrderTotal}
```

And now we can test it:

```
let zero = EmptyOrder

// test identity
let productLine = Product {ProductCode="AAA"; Qty=2; Price=9.99; LineTotal=19.98}
assert (productLine = addLine productLine zero)
assert (productLine = addLine zero productLine)

let totalLine = Total {Qty=2; OrderTotal=19.98}
assert (totalLine = addLine totalLine zero)
assert (totalLine = addLine zero totalLine)
```

Using the built in List.sum function

It turns out that the `List.sum` function knows about monoids! If you tell it what the addition operation is, and what the zero is, then you can use `List.sum` directly rather than `List.fold`.

The way you do this is by attaching two static members, `+` and `Zero` to your type, like this:

```
type OrderLine with
    static member (+) (x,y) = addLine x y
    static member Zero = EmptyOrder   // a property
```

Once this has been done, you can use `List.sum` and it will work as expected.

```
let lines1 = [productLine]
// using fold with explicit op and zero
lines1 |> List.fold addLine zero |> printfn "%A"
// using sum with implicit op and zero
lines1 |> List.sum |> printfn "%A"

let emptyList: OrderLine list = []
// using fold with explicit op and zero
emptyList |> List.fold addLine zero |> printfn "%A"
// using sum with implicit op and zero
emptyList |> List.sum |> printfn "%A"
```

Note that for this to work you mustn't already have a method or case called `Zero`. If I had used the name `Zero` instead of `EmptyOrder` for the third case it would not have worked.

Although this is a neat trick, in practice I don't think it is a good idea unless you are defining a proper math-related type such as `ComplexNumber` or `Vector`. It's a bit too clever and non-obvious for my taste.

If you *do* want to use this trick, your `Zero` member cannot be an extension method -- it must be defined with the type.

For example, in the code below, I'm trying to define the empty string as the "zero" for strings.

`List.fold` works because `String.Zero` is visible as an extension method in this code right here, but `List.sum` fails because the extension method is not visible to it.

```
module StringMonoid =

    // define extension method
    type System.String with
        static member Zero = ""

    // OK.
    ["a";"b";"c"]
    |> List.reduce (+)
    |> printfn "Using reduce: %s"

    // OK. String.Zero is visible as an extension method
    ["a";"b";"c"]
    |> List.fold (+) System.String.Zero
    |> printfn "Using fold: %s"

    // Error. String.Zero is NOT visible to List.sum
    ["a";"b";"c"]
    |> List.sum
    |> printfn "Using sum: %s"
```

Mapping to a different structure

Having two different cases in a union might be acceptable in the order line case, but in many real world cases, that approach is too complicated or confusing.

Consider a customer record like this:

```
open System

type Customer = {
    Name:string // and many more string fields!
    LastActive:DateTime
    TotalSpend:float }
```

How would we "add" two of these customers?

A helpful tip is to realize that aggregation really only works for numeric and similar types. Strings can't really be aggregated easily.

So rather than trying to aggregate `Customer`, let's define a separate class `CustomerStats` that contains all the aggregatable information:

```
// create a type to track customer statistics
type CustomerStats = {
    // number of customers contributing to these stats
    Count:int
    // total number of days since last activity
    TotalInactiveDays:int
    // total amount of money spent
    TotalSpend:float }
```

All the fields in `CustomerStats` are numeric, so it is obvious how we can add two stats together:

```
let add stat1 stat2 = {
    Count = stat1.Count + stat2.Count;
    TotalInactiveDays = stat1.TotalInactiveDays + stat2.TotalInactiveDays
    TotalSpend = stat1.TotalSpend + stat2.TotalSpend
    }

// define an infix version as well
let (++) a b = add a b
```

As always, the inputs and output of the `add` function must be the same type. We must have `CustomerStats -> CustomerStats -> CustomerStats`, not `Customer -> Customer -> CustomerStats` or any other variant.

Ok, so far so good.

Now let's say we have a collection of customers, and we want to get the aggregated stats for them, how should we do this?

We can't add the customers directly, so what we need to do is first convert each customer to a `CustomerStats`, and then add the stats up using the monoid operation.

Here's an example:

```
// convert a customer to a stat
let toStats cust =
    let inactiveDays= DateTime.Now.Subtract(cust.LastActive).Days;
    {Count=1; TotalInactiveDays=inactiveDays; TotalSpend=cust.TotalSpend}

// create a list of customers
let c1 = {Name="Alice"; LastActive=DateTime(2005,1,1); TotalSpend=100.0}
let c2 = {Name="Bob"; LastActive=DateTime(2010,2,2); TotalSpend=45.0}
let c3 = {Name="Charlie"; LastActive=DateTime(2011,3,3); TotalSpend=42.0}
let customers = [c1;c2;c3]

// aggregate the stats
customers
|> List.map toStats
|> List.reduce add
|> printfn "result = %A"
```

The first thing to note is that the `toStats` creates statistics for just one customer. We set the count to just 1. It might seem a bit strange, but it does make sense, because if there was just one customer in the list, that's what the aggregate stats would be.

The second thing to note is how the final aggregation is done. First we use `map` to convert the source type to a type that is a monoid, and then we use `reduce` to aggregate all the stats.

Hmmm.... `map` followed by `reduce`. Does that sound familiar to you?

Yes indeed, Google's famous MapReduce algorithm was inspired by this concept (although the details are somewhat different).

Before we move on, here are some simple exercises for you to check your understanding.

- What is the "zero" for `CustomerStats`? Test your code by using `List.fold` on an empty list.

- Write a simple `OrderStats` class and use it to aggregate the `OrderLine` type that we introduced at the beginning of this post.

Monoid Homomorphisms

We've now got all the tools we need to understand something called a *monoid homomorphism*.

I know what you're thinking... Ugh! Not just one, but two strange math words at once!

But I hope that the word "monoid" is not so intimidating now. And "homomorphism" is another math word that is simpler than it sounds. It's just greek for "same shape" and it describes a mapping or function that keeps the "shape" the same.

What does that mean in practice?

Well, we have seen that all monoids have a certain common structure. That is, even though the underlying objects can be quite different (integers, strings, lists, `CustomerStats`, etc.) the "monoidness" of them is the same. As George W. Bush once said, once you've seen one monoid, you've seen them all.

So a *monoid* homomorphism is a transformation that preserves an essential "monoidness", even if the "before" and "after" objects are quite different.

In this section, we'll look at a simple monoid homomorphism. It's the "hello world", the "fibonacci series", of monoid homomorphisms -- word counting.

Documents as a monoid

Let's say we have a type which represents text blocks, something like this:

```
type Text = Text of string
```

And of course we can add two smaller text blocks to make a larger text block:

```
let addText (Text s1) (Text s2) =
    Text (s1 + s2)
```

Here's an example of how adding works:

```
let t1 = Text "Hello"
let t2 = Text " World"
let t3 = addText t1 t2
```

Since you are now a expert, you will quickly recognize this as a monoid, with the zero obviously being `Text ""`.

Now let's say we are writing a book (such as this one) and we want a word count to show how much we have written.

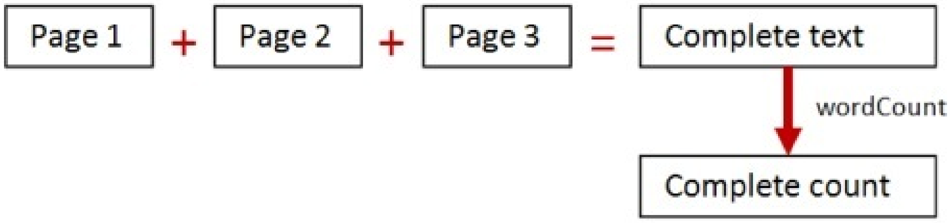
Here's a very crude implementation, plus a test:

```
let wordCount (Text s) =
    s.Split(' ').Length

// test
Text "Hello world"
|> wordCount
|> printfn "The word count is %i"
```

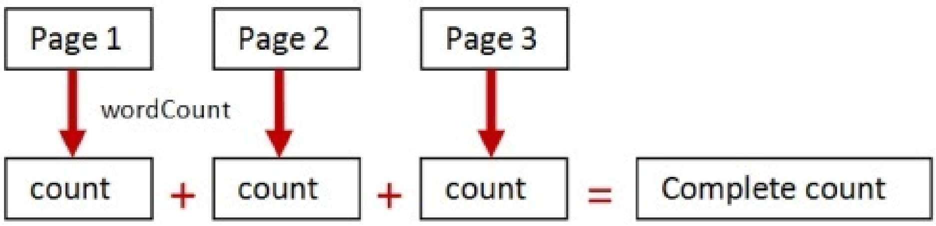
So we are writing away, and now we have produced three pages of text. How do we calculate the word count for the complete document?

Well, one way is to add the separate pages together to make a complete text block, and then apply the `wordCount` function to that text block. Here's a diagram:



But everytime we finish a new page, we have to add all the text together and do the word count all over again.

No doubt you can see that there is a better way of doing this. Instead of adding all the text together and then counting, get the word count for each page separately, and then add these counts up, like this:



The second approach relies on the fact that integers (the counts) are themselves a monoid, and you can add them up to get the desired result.

So the `wordCount` function has transformed an aggregation over "pages" into an aggregation over "counts".

The big question now: is `wordCount` a monoid homomorphism?

Well, pages (text) and counts (integers) are both monoids, so it certainly transforms one monoid into another.

But the more subtle condition is: does it preserve the "shape"? That is, does the adding of the counts give the same answer as the adding of the pages?

In this case, the answer is yes. So `wordCount` *is* a monoid homomorphism!

You might think that this is obvious, and that all mappings like this must be monoid homomorphisms, but we'll see an example later where this is not true.

The benefits of chunkability

The advantage of the monoid homomorphism approach is that it is *"chunkable"*.

Each map and word count is independent of the others, so we can do them separately and then add up the answers afterwards. For many algorithms, working on small chunks of data is much more efficient than working on large chunks, so if we can, we should exploit this whenever possible.

As a direct consequence of this chunkability, we get some of the benefits that we touched on in the previous post.

First, it is *incremental*. That is, as we add text to the last page, we don't have to recalculate the word counts for all the previous pages, which might save some time.

Second, it is *parallelizable*. The work for each chunk can be done independently, on different cores or machines. Note that in practice, parallelism is much overrated. The chunkability into small pieces has a much greater effect on performance than parallelism itself.

Comparing word count implementations

We're now ready to create some code that will demonstrate these two different techniques.

Let's start with the basic definitions from above, except that I will change the word count to use regular expressions instead of `split`.

```
module WordCountTest =
    open System

    type Text = Text of string

    let addText (Text s1) (Text s2) =
        Text (s1 + s2)

    let wordCount (Text s) =
        System.Text.RegularExpressions.Regex.Matches(s,@"\S+").Count
```

Next, we'll create a page with 1000 words in it, and a document with 1000 pages.

```
module WordCountTest =

    // code as above

    let page() =
        List.replicate 1000 "hello "
        |> List.reduce (+)
        |> Text

    let document() =
        page() |> List.replicate 1000
```

We'll want to time the code to see if there is any difference between the implementations. Here's a little helper function.

```
module WordCountTest =

    // code as above

    let time f msg =
        let stopwatch = Diagnostics.Stopwatch()
        stopwatch.Start()
        f()
        stopwatch.Stop()
        printfn "Time taken for %s was %ims" msg stopwatch.ElapsedMilliseconds
```

Ok, let's implement the first approach. We'll add all the pages together using `addText` and then do a word count on the entire million word document.

```
module WordCountTest =

    // code as above

    let wordCountViaAddText() =
        document()
        |> List.reduce addText
        |> wordCount
        |> printfn "The word count is %i"

    time wordCountViaAddText "reduce then count"
```

For the second approach, we'll do `wordCount` on each page first, and then add all the results together (using `reduce` of course).

```
module WordCountTest =

    // code as above

    let wordCountViaMap() =
        document()
        |> List.map wordCount
        |> List.reduce (+)
        |> printfn "The word count is %i"

    time wordCountViaMap "map then reduce"
```

Note that we have only changed two lines of code!

In `wordCountViaAddText` we had:

```
|> List.reduce addText
|> wordCount
```

And in `wordCountViaMap` we have basically swapped these lines. We now do `wordCount` *first* and then `reduce` afterwards, like this:

```
|> List.map wordCount
|> List.reduce (+)
```

Finally, let's see what difference parallelism makes. We'll use the built-in `Array.Parallel.map` instead of `List.map`, which means we'll need to convert the list into an array first.

```
module WordCountTest =

    // code as above

    let wordCountViaParallelAddCounts() =
        document()
        |> List.toArray
        |> Array.Parallel.map wordCount
        |> Array.reduce (+)
        |> printfn "The word count is %i"

    time wordCountViaParallelAddCounts "parallel map then reduce"
```

I hope that you are following along with the implementations, and that you understand what is going on.

Analyzing the results

Here are the results for the different implementations running on my 4 core machine:

```
Time taken for reduce then count was 7955ms
Time taken for map then reduce was 698ms
Time taken for parallel map then reduce was 603ms
```

We must recognize that these are crude results, not a proper performance profile. But even so, it is very obvious that the map/reduce version is about 10 times faster that the `ViaAddText` version.

This is the key to why monoid homomorphisms are important -- they enable a "divide and conquer" strategy that is both powerful and easy to implement.

Yes, you could argue that the algorithms used are very inefficient. String concat is a terrible way to accumulate large text blocks, and there are much better ways of doing word counts. But even with these caveats, the fundamental point is still valid: by swapping two lines of code, we got a huge performance increase.

And with a little bit of hashing and caching, we would also get the benefits of incremental aggregation -- only recalculating the minimum needed as pages change.

Note that the parallel map didn't make that much difference in this case, even though it did use all four cores. Yes, we did add some minor expense with `toArray` but even in the best case, you might only get a small speed up on a multicore machine. To reiterate, what really made the most difference was the divide and conquer strategy inherent in the map/reduce approach.

A non-monoid homomorphism

I mentioned earlier that not all mappings are necessarily monoid homomorphisms. In this section, we'll look at an example of one that isn't.

For this example, rather than using counting words, we're going to return the most frequent word in a text block.

Here's the basic code.

```
module FrequentWordTest =

    open System
    open System.Text.RegularExpressions

    type Text = Text of string

    let addText (Text s1) (Text s2) =
        Text (s1 + s2)

    let mostFrequentWord (Text s) =
        Regex.Matches(s,@"\S+")
        |> Seq.cast<Match>
        |> Seq.map (fun m -> m.ToString())
        |> Seq.groupBy id
        |> Seq.map (fun (k,v) -> k,Seq.length v)
        |> Seq.sortBy (fun (_,v) -> -v)
        |> Seq.head
        |> fst
```

The `mostFrequentWord` function is bit more complicated than the previous `wordCount` function, so I'll take you through it step by step.

First, we use a regex to match all non-whitespace. The result of this is a `MatchCollection` not a list of `Match`, so we have to explicitly cast it into a sequence (an `IEnumerable<Match>` in C# terms).

Next we convert each `Match` into the matched word, using `ToString()`. Then we group by the word itself, which gives us a list of pairs, where each pair is a `(word,list of words)`. We then turn those pairs into `(word,list count)` and then sort descending (using the negated word count).

Finally we take the first pair, and return the first part of the pair. This is the most frequent word.

Ok, let's continue, and create some pages and a document as before. This time we're not interested in performance, so we only need a few pages. But we do want to create *different* pages. We'll create one containing nothing but "hello world", another containing nothing but "goodbye world", and a third containing "foobar". (Not a very interesting book IMHO!)

```
module FrequentWordTest =

    // code as above

    let page1() =
        List.replicate 1000 "hello world "
        |> List.reduce (+)
        |> Text

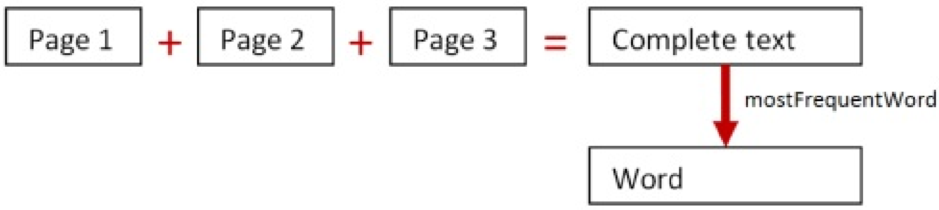
    let page2() =
        List.replicate 1000 "goodbye world "
        |> List.reduce (+)
        |> Text

    let page3() =
        List.replicate 1000 "foobar "
        |> List.reduce (+)
        |> Text

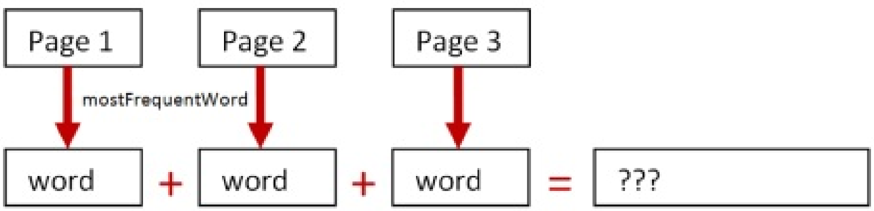
    let document() =
        [page1(); page2(); page3()]
```

It is obvious that, with respect to the entire document, "world" is the most frequent word overall.

So let's compare the two approaches as before. The first approach will combine all the pages and then apply `mostFrequentWord`, like this.



The second approach will do `mostFrequentWord` separately on each page and then combine the results, like this:



Here's the code:

```
module FrequentWordTest =

    // code as above

    document()
    |> List.reduce addText
    |> mostFrequentWord
    |> printfn "Using add first, the most frequent word is %s"

    document()
    |> List.map mostFrequentWord
    |> List.reduce (+)
    |> printfn "Using map reduce, the most frequent word is %s"
```

Can you see what happened? The first approach was correct. But the second approach gave a completely wrong answer!

```
Using add first, the most frequent word is world
Using map reduce, the most frequent word is hellogoodbyefoobar
```

The second approach just concatenated the most frequent words from each page. The result is a new string that was not on *any* of the pages. A complete fail!

What went wrong?

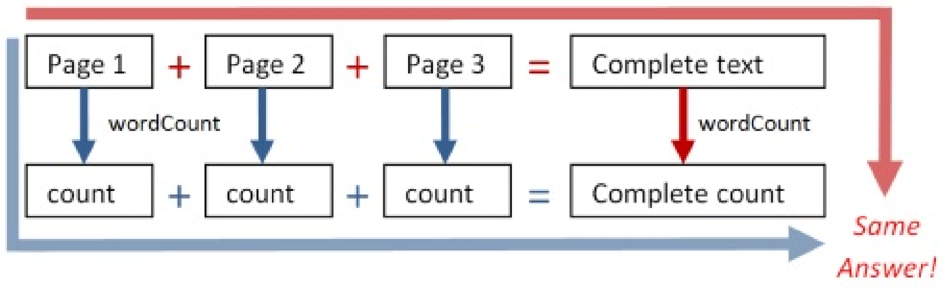
Well, strings *are* a monoid under concatenation, so the mapping transformed a monoid (Text) to another monoid (string).

But the mapping did not preserve the "shape". The most frequent word in a big chunk of text cannot be derived from the most frequent words in smaller chunks of text. In other words, it is not a proper monoid homomorphism.

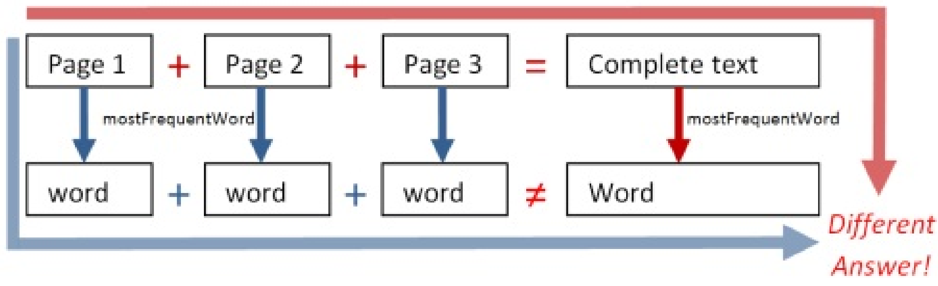
Definition of a monoid homomorphism

Let's look at these two different examples again to understand what the distinction is between them.

In the word count example, we got the *same* final result whether we added the blocks and then did the word count, or whether we did the word counts and then added them together. Here's a diagram:



But for the most frequent word example, we did *not* get the same answer from the two different approaches.



In other words, for `wordCount`, we had

```
wordCount(page1) + wordCount(page2) EQUALS wordCount(page1 + page)
```

But for `mostFrequentWord`, we had:

```
mostFrequentWord(page1) + mostFrequentWord(page2) NOT EQUAL TO mostFrequentWord(page1
+ page)
```

So this brings us to a slightly more precise definition of a monoid homomorphism:

```
Given a function that maps from one monoid to another (like 'wordCount' or 'mostFreque
ntWord')

Then to be a monoid homomorphism, the function must meet the requirement that:

function(chunk1) + function(chunk2) MUST EQUAL function(chunk1 + chunk2)
```

Alas, then, `mostFrequentWord` is not a monoid homomorphism.

That means that if we want to calculate the `mostFrequentWord` on a large number of text files, we are sadly forced to add all the text together first, and we can't benefit from a divide and conquer strategy.

... or can we? Is there a way to turn `mostFrequentWord` into a proper monoid homomorphism? Stay tuned!

Next steps

So far, we have only dealt with things that are proper monoids. But what if the thing you want to work with is *not* a monoid? What then?

In the next post in this series, I'll give you some tips on converting almost anything into a monoid.

We'll also fix up the `mostFrequentWord` example so that it is a proper monoid homomorphism, and we'll revisit the thorny problem of zeroes, with an elegant approach for creating them.

See you then!

Further reading

If you are interested in using monoids for data aggregation, there are lots of good discussions in the following links:

- Twitter's Algebird library
- Most probabilistic data structures are monoids.
- Gaussian distributions form a monoid.
- Google's MapReduce Programming Model (PDF).
- Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms (PDF).
- LinkedIn's Hourglass libary for Hadoop
- From Stack Exchange: What use are groups, monoids, and rings in database computations?

If you want to get a bit more technical, here is a detailed study of monoids and semigroups, using graphics diagrams as the domain:

- Monoids: Theme and Variations (PDF).

Working with non-monoids

In the previous posts in this series, we only dealt with things that were proper monoids.

But what if the thing you want to work with is *not* a monoid? What then? Well, in this post, I'll give you some tips on converting almost anything into a monoid.

In the process, we'll be introduced to a number of important and common functional design idioms, such as preferring lists rather than singletons, and using the option type at every opportunity.

Getting closure

If you recall, for a proper monoid, we need three things to be true: closure, associativity, and identity. Each requirement can present a challenge, so we'll discuss each in turn.

We'll start with closure.

In some cases you might want to add values together, but the type of the combined value is not the same as the type of the original values. How can you handle this?

One way is to just to map from the original type to a new type that *is* closed. We saw this approach used with the `Customer` and `CustomerStats` example in the previous post. In many cases, this is the easiest approach, because you don't have to mess with the design of the original types.

On the other hand, sometimes you really don't want to use `map`, but instead want to design your type from the beginning so that it meets the closure requirement.

Either way, whether you are designing a new type or redesigning an existing type, you can use similar techniques to get closure.

Combining closed types to make new compound types

Obviously, we've seen that numeric types are closed under some basic math operations like addition and multiplication. We've also seen that some non-numeric types, like strings and lists, are closed under concatenation.

With this in mind, it should be obvious that any combination of these types will be closed too. We just have to define the "add" function to do the appropriate "add" on the component types.

Here's an example:

```
type MyType = {count:int; items:int list}

let addMyType t1 t2 =
    {count = t1.count + t2.count;
     items = t1.items @ t2.items}
```

The `addMyType` function uses integer addition on the `int` field, and list concatenation on the `list` field. As a result the `MyType` is closed using the function `addMyType` -- in fact, not only is it closed, it is a monoid too. So in this case, we're done!

This is exactly the approach we took with `CustomerStats` in the previous post.

So here's my first tip:

- **DESIGN TIP: To easily create a monoidal type, make sure that each field of the type is also a monoid.**

Question to think about: when you do this, what is the "zero" of the new compound type?

Dealing with non-numeric types

The approach above works when creating compound types. But what about non-numeric types, which have no obvious numeric equivalent?

Here's a very simple case. Say that you have some chars that you want to add together, like this:

```
'a' + 'b' -> what?
```

But, a char plus a char is not another char. If anything, it is a string.

```
'a' + 'b' -> "ab" // Closure fail!
```

But that is very unhelpful, as it does not meet the closure requirement.

One way to fix this is to force the chars into strings, which does work:

```
"a" + "b" -> "ab"
```

But that is a specific solution for chars -- is there a more generic solution that will work for other types?

Well, think for a minute what the relationship of a `string` to a `char` is. A string can be thought of as a list or array of chars.

In other words, we could have used lists of chars instead, like this:

```
['a'] @ ['b'] -> ['a'; 'b'] // Lists FTW!
```

This meets the closure requirement as well.

What's more, this is in fact a general solution to *any* problem like this, because *anything* can be put into a list, and lists (with concatenation) are always monoids.

So here's my next tip:

- **DESIGN TIP: To enable closure for a non-numeric type, replace single items with lists.**

In some cases, you might need to convert to a list when setting up the monoid and then convert to another type when you are done.

For example, in the `Char` case, you would do all your manipulation on lists of chars and then only convert to a string at the end.

So let's have a go at creating a "monoidal char" module.

```
module MonoidalChar =
    open System

    /// "monoidal char"
    type MChar = MChar of Char list

    /// convert a char into a "monoidal char"
    let toMChar ch = MChar [ch]

    /// add two monoidal chars
    let addChar (MChar l1) (MChar l2) =
        MChar (l1 @ l2)

    // infix version
    let (++) = addChar

    /// convert to a string
    let toString (MChar cs) =
        new System.String(List.toArray cs)
```

You can see that `MChar` is a wrapper around a list of chars, rather than a single char.

Now let's test it:

```
open MonoidalChar

// add two chars and convert to string
let a = 'a' |> toMChar
let b = 'b' |> toMChar
let c = a ++ b
c |> toString |> printfn "a + b = %s"
// result: "a + b = ab"
```

If we want to get fancy we can use map/reduce to work on a set of chars, like this:

```
[' '..'z']   // get a lot of chars
|> List.filter System.Char.IsPunctuation
|> List.map toMChar
|> List.reduce addChar
|> toString
|> printfn "punctuation chars are %s"
// result: "punctuation chars are !"#%&'()*,-./:;?@[\]_"
```

Monoids for errors

The `MonoidalChar` example is trivial, and could perhaps be implemented in other ways, but in general this is an extremely useful technique.

For example, here is a simple module for doing some validation. There are two options, `Success` and `Failure`, and the `Failure` case also has a error string associated with it.

```
module Validation =

    type ValidationResult =
        | Success
        | Failure of string

    let validateBadWord badWord (name:string) =
        if name.Contains(badWord) then
            Failure ("string contains a bad word: " + badWord)
        else
            Success

    let validateLength maxLength name =
        if String.length name > maxLength then
            Failure "string is too long"
        else
            Success
```

In practice, we might perform multiple validations on a string, and we would like to return all the results at once, added together somehow.

This calls out for being a monoid! If we can add two results pairwise, then we can extend the operation to add as many results as we like!

So then the question is, how do we combine *two* validation results?

```
let result1 = Failure "string is null or empty"
let result2 = Failure "string is too long"

result1 + result2 = ????
```

A naive approach would be to concatenate the strings, but that wouldn't work if we were using format strings, or resource ids with localization, etc.

No, a better way is to convert the `Failure` case to use a *list* of strings instead of a single string. That will make combining results simple.

Here's the same code as above, with the `Failure` case redefined to use a list:

```
module MonoidalValidation =

    type ValidationResult =
        | Success
        | Failure of string list

    // helper to convert a single string into the failure case
    let fail str =
        Failure [str]

    let validateBadWord badWord (name:string) =
        if name.Contains(badWord) then
            fail ("string contains a bad word: " + badWord)
        else
            Success

    let validateLength maxLength name =
        if String.length name > maxLength then
            fail "string is too long"
        else
            Success
```

You can see that the individual validations call `fail` with a single string, but behind the scenes it is being stored as a list of strings, which can, in turn, be concatenated together.

With this in place, we can now create the `add` function.

The logic will be:

- If both results are `Success`, then the combined result is `Success`

- If one result is `Failure`, then the combined result is that failure.
- If both results are `Failure`, then the combined result is a `Failure` with both error lists concatenated.

Here's the code:

```
module MonoidalValidation =

    // as above

    /// add two results
    let add r1 r2 =
        match r1,r2 with
        | Success,    Success -> Success
        | Failure f1, Success -> Failure f1
        | Success,    Failure f2 -> Failure f2
        | Failure f1, Failure f2 -> Failure (f1 @ f2)
```

Here are some tests to check the logic:

```
open MonoidalValidation

let test1 =
    let result1 = Success
    let result2 = Success
    add result1 result2
    |> printfn "Result is %A"
    // "Result is Success"

let test2 =
    let result1 = Success
    let result2 = fail "string is too long"
    add result1 result2
    |> printfn "Result is %A"
    // "Result is Failure ["string is too long"]"

let test3 =
    let result1 = fail "string is null or empty"
    let result2 = fail "string is too long"
    add result1 result2
    |> printfn "Result is %A"

    // Result is Failure
    //   [ "string is null or empty";
    //     "string is too long"]
```

And here's a more realistic example, where we have a list of validation functions that we want to apply:

```
let test4 =
    let validationResults str =
        [
        validateLength 10
        validateBadWord "monad"
        validateBadWord "cobol"
        ]
        |> List.map (fun validate -> validate str)

    "cobol has native support for monads"
    |> validationResults
    |> List.reduce add
    |> printfn "Result is %A"
```

The output is a `Failure` with three error messages.

```
Result is Failure
  ["string is too long"; "string contains a bad word: monad";
   "string contains a bad word: cobol"]
```

One more thing is needed to finish up this monoid. We need a "zero" as well. What should it be?

By definition, it is something that when combined with another result, leaves the other result alone.

I hope you can see that by this definition, "zero" is just `Success`.

```
module MonoidalValidation =

    // as above

    // identity
    let zero = Success
```

As you know, we would need to use zero if the list to reduce over is empty. So here's an example where we don't apply any validation functions at all, giving us an empty list of `ValidationResult`.

```
let test5 =
    let validationResults str =
        []
        |> List.map (fun validate -> validate str)

    "cobol has native support for monads"
    |> validationResults
    |> List.fold add zero
    |> printfn "Result is %A"

    // Result is Success
```

Note that we needed to change `reduce` to `fold` as well, otherwise we would get a runtime error.

Lists for performance

Here's one more example of the benefit of using lists. Compared with other methods of combination, list concatenation is relatively cheap, both in computation and in memory use, because the objects being pointed to don't have to change or be reallocated.

For example, in the previous post, we defined a `Text` block that wrapped a string, and used string concatenation to add their contents.

```
type Text = Text of string

let addText (Text s1) (Text s2) =
    Text (s1 + s2)
```

But for large strings this continual concatenation can be expensive.

Consider a different implementation, where the `Text` block contains a *list* of strings instead.

```
type Text = Text of string list

let addText (Text s1) (Text s2) =
    Text (s1 @ s2)
```

Almost no change in implementation, but performance will probably be greatly improved.

You can do all your manipulation on *lists* of strings and you need only convert to a normal string at the very end of the processing sequence.

And if lists aren't performant enough for you, you can easily extend this approach to use classic data structures like trees, heaps, etc. or mutable types like ResizeArray. (See the appendix on performance at the bottom of this post for some more discussion on this)

Jargon alert

The concept of using a list of objects as a monoid is common in mathematics, where it is called a "free monoid". In computer science, it also called a "Kleene star" such as `A*`. And if you don't allow empty lists, then you have no zero element. This variant is called a "free semigroup" or "Kleene plus" such as `A+`.

This "star" and "plus" notation will surely be familiar to you if you have ever used regular expressions.*

* You probably weren't aware that there was a connection between regular expressions and monoids! There's some even deeper relationships too.

Associativity

Now that we have dealt with closure, let's take on associativity.

We saw a couple of non-associative operations in the very first post, including subtraction and division.

We can see that `5 - (3 - 2)` is not equal to `(5 - 3) - 2`. This shows that subtraction is not associative, and also `12 / (3 / 2)` is not equal to `(12 / 3) / 2`, which shows that division is not associative.

There's no single correct answer in these cases, because you might genuinely care about different answers depending on whether you work from left to right or right to left.

In fact, the F# standard libraries have two versions of `fold` and `reduce` to cater for each preference. The normal `fold` and `reduce` work left to right, like this:

```
//same as (12 - 3) - 2
[12;3;2] |> List.reduce (-)  // => 7

//same as ((12 - 3) - 2) - 1
[12;3;2;1] |> List.reduce (-)  // => 6
```

But there is also `foldBack` and `reduceBack` that work from right to left, like this:

```
//same as 12 - (3 - 2)
[12;3;2] |> List.reduceBack (-) // => 11

//same as 12 - (3 - (2 - 1))
[12;3;2;1] |> List.reduceBack (-) // => 10
```

In a sense, then, the associativity requirement is just a way of saying that you should get the *same* answer no matter whether you use `fold` or `foldBack`.

Moving the operation into the element

But assuming that you *do* want a consistent monoidal approach, the trick in many cases is to move the operation into a property of each element. **Make the operation a noun, rather than a verb.**

For example `3 - 2` can be thought of as `3 + (-2)`. Rather than "subtraction" as a verb, we have "negative 2" as a noun.

In that case, the above example becomes `5 + (-3) + (-2)`. And since we are now using addition as the operator, we *do* have associativity, and `5 + (-3 + -2)` is indeed the same as `(5 + -3) + -2`.

A similar approach works with division. `12 / 3 / 2` can be converted into `12 * (1/3) * (1/2)`, and now we are back to multiplication as the operator, which is associative.

This approach of converting the operator into a property of the element can be generalized nicely.

So here's a tip:

- **DESIGN TIP: To get associativity for an operation, try to move the operation into the object.**

We can revisit an earlier example to understand how this works. If you recall, in the first post we tried to come up with a non-associative operation for strings, and settled on `subtractChars`.

Here's a simple implementation of `subtractChars`

```
let subtractChars (s1:string) (s2:string) =
    let isIncluded (ch:char) = s2.IndexOf(ch) = -1
    let chars = s1.ToCharArray() |> Array.filter isIncluded
    System.String(chars)

// infix version
let (--) = subtractChars
```

With this implementation we can do some interactive tests:

```
"abcdef" -- "abd"   //  "cef"
"abcdef" -- ""      //  "abcdef"
```

And we can see for ourselves that the associativity requirement is violated:

```
("abc" -- "abc") -- "abc"  // ""
"abc" -- ("abc" -- "abc")  // "abc"
```

How can we make this associative?

The trick is move the "subtract-ness" from the operator into the object, just as we did with the numbers earlier.

What I mean is that we replace the plain strings with a "subtract" or "chars to remove" data structure that captures what we want to remove, like so:

```
let removalAction = (subtract "abd") // a data structure
```

And then we "apply" the data structure to the string:

```
let removalAction = (subtract "abd")
removalAction |> applyTo "abcdef"  // "Result is cef"
```

Once we use this approach, we can rework the non-associative example above to look something like this:

```
let removalAction = (subtract "abc") + (subtract "abc") + (subtract "abc")
removalAction |> applyTo "abc"    // "Result is "
```

Yes, it is not exactly the same as the original code, but you might find that this is actually a better fit in many situations.

The implementation is below. We define a `CharsToRemove` to contain a set of chars, and the other function implementations fall out from that in a straightforward way.

```
/// store a list of chars to remove
type CharsToRemove = CharsToRemove of Set<char>

/// construct a new CharsToRemove
let subtract (s:string) =
    s.ToCharArray() |> Set.ofArray |>  CharsToRemove

/// apply a CharsToRemove to a string
let applyTo (s:string) (CharsToRemove chs) =
    let isIncluded ch = Set.exists ((=) ch) chs |> not
    let chars = s.ToCharArray() |> Array.filter isIncluded
    System.String(chars)

// combine two CharsToRemove to get a new one
let (++) (CharsToRemove c1) (CharsToRemove c2) =
    CharsToRemove (Set.union c1 c2)
```

Let's test!

```
let test1 =
    let removalAction = (subtract "abd")
    removalAction |> applyTo "abcdef" |> printfn "Result is %s"
    // "Result is cef"

let test2 =
    let removalAction = (subtract "abc") ++ (subtract "abc") ++ (subtract "abc")
    removalAction |> applyTo "abcdef" |> printfn "Result is %s"
    // "Result is "
```

The way to think about this approach is that, in a sense, we are modelling *actions* rather than *data*. We have a list of `CharsToRemove` actions, then we combine them into a single "big" `CharsToRemove` action, and then we execute that single action at the end, after we have finished the intermediate manipulations.

We'll see another example of this shortly, but you might be thinking at this point: "this sounds a bit like functions, doesn't it?" To which I will say "yes, it does!"

In fact rather than creating this `CharsToRemove` data structure, we could have just partially applied the original `subtractChars` function, as shown below:

(Note that we reverse the parameters to make partial application easier)

```
// reverse for partial application
let subtract str charsToSubtract =
    subtractChars charsToSubtract str

let removalAction = subtract "abd"
"abcdef" |> removalAction |> printfn "Result is %s"
// "Result is cef"
```

And now we don't even need a special `applyTo` function.

But when we have more than one of these subtraction functions, what do we do? Each of these partially applied functions has signature `string -> string`, so how can we "add" them together?

```
(subtract "abc") + (subtract "abc") + (subtract "abc")  = ?
```

The answer is function composition, of course!

```
let removalAction2 = (subtract "abc") >> (subtract "abc") >> (subtract "abc")
removalAction2 "abcdef" |> printfn "Result is %s"
// "Result is def"
```

This is the functional equivalent of creating the `CharsToRemove` data structure.

The "data structure as action" and function approach are not exactly the same -- the `CharsToRemove` approach may be more efficient, for example, because it uses a set, and is only applied to strings at the end -- but they both achieve the same goal. Which one is better depends on the particular problem you're working on.

I'll have more to say on functions and monoids in the next post.

Identity

Now to the last requirement for a monoid: identity.

As we have seen, identity is not always needed, but it is nice to have if you might be dealing with empty lists.

For numeric values, finding an identity for an operation is generally easy, whether it be `0` (addition), `1` (multiplication) or `Int32.MinValue` (max).

And this carries over to structures that contain only numeric values as well -- just set all values to their appropriate identity. The `CustomerStats` type from the previous post demonstrates that nicely.

But what if you have objects that are not numeric? How can you create a "zero" or identity element if there is no natural candidate?

The answer is: *you just make one up*.

Seriously!

We have already seen an example of this in the previous post, when we added an `EmptyOrder` case to the `OrderLine` type:

```
type OrderLine =
    | Product of ProductLine
    | Total of TotalLine
    | EmptyOrder
```

Let's look at this more closely. We performed two steps:

- First, we created a new case and added it to the list of alternatives for an `OrderLine` (as shown above).
- Second, we adjusted the `addLine` function to take it into account (as shown below).

```
let addLine orderLine1 orderLine2 =
    match orderLine1,orderLine2 with
    // is one of them zero? If so, return the other one
    | EmptyOrder, _ -> orderLine2
    | _, EmptyOrder -> orderLine1
    // logic for other cases ...
```

That's all there is to it.

The new, augmented type consists of the old order line cases, *plus* the new `EmptyOrder` case, and so it can reuse much of the behavior of the old cases.

In particular, can you see that the new augmented type follows all the monoid rules?

- A pair of values of the new type can be added to get another value of the new type (closure)
- If the combination order didn't matter for the old type, then it still doesn't matter for the new type (associativity)
- And finally... this extra case now gives us an identity for the new type.

Turning PositiveNumber into a monoid

We could do the same thing with the other semigroups we've seen.

For example, we noted earlier that strictly positive numbers (under addition) didn't have an identity; they are only a semigroup. If we wanted to create a zero using the "augmentation with extra case" technique (rather than just using `0` !) we would first define a special `Zero` case (not an integer), and then create an `addPositive` function that can handle it, like this:

```
type PositiveNumberOrIdentity =
    | Positive of int
    | Zero

let addPositive i1 i2 =
    match i1,i2 with
    | Zero, _ -> i2
    | _, Zero -> i1
    | Positive p1, Positive p2 -> Positive (p1 + p2)
```

Admittedly, `PositiveNumberOrIdentity` is a contrived example, but you can see how this same approach would work for any situation where you have "normal" values and a special, separate, zero value.

A generic solution

There are a few drawbacks to this:

- We have to deal with *two* cases now: the normal case and the zero case.
- We have to create custom types and custom addition functions

Unfortunately, there's nothing you can do about the first issue. If you have a system with no natural zero, and you create an artificial one, then you will indeed always have to deal with two cases.

But there *is* something you can do about the second issue! Rather than create a new custom type over and over, perhaps can we create a *generic* type that has two cases: one for all normal values and one for the artificial zero, like this:

```
type NormalOrIdentity<'T> =
    | Normal of 'T
    | Zero
```

Does this type look familiar? It's just the **Option type** in disguise!

In other words, any time we need an identity which is outside the normal set of values, we can use `Option.None` to represent it. And then `Option.Some` is used for all the other "normal" values.

Another benefit of using `Option` is that we can also write a completely generic "add" function as well. Here's a first attempt:

```
let optionAdd o1 o2 =
    match o1, o2 with
    | None, _ -> o2
    | _, None -> o1
    | Some s1, Some s2 -> Some (s1 + s2)
```

The logic is straightforward. If either option is `None`, the other option is returned. If both are `Some`, then they are unwrapped, added together, and then wrapped in a `Some` again.

But the `+` in the last line makes assumptions about the types that we are adding. Better to pass in the addition function explicitly, like this:

```
let optionAdd f o1 o2 =
    match o1, o2 with
    | None, _ -> o2
    | _, None -> o1
    | Some s1, Some s2 -> Some (f s1 s2)
```

In practice, this would used with partial application to bake in the addition function.

So now we have another important tip:

- **DESIGN TIP: To get identity for an operation, create a special case in a discriminated union, or, even simpler, just use Option.**

PositiveNumber revisited

So here is the Positive Number example again, now using the `Option` type.

```
type PositiveNumberOrIdentity = int option
let addPositive = optionAdd (+)
```

Much simpler!

Notice that we pass in the "real" addition function as a parameter to `optionAdd` so that it is baked in. In other situations, you would do the same with the relevant aggregation function that is associated with the semigroup.

As a result of this partial application, `addPositive` has the signature: `int option -> int option -> int option`, which is exactly what we would expect from a monoid addition function.

In other words, `optionAdd` turns any function `'a -> 'a -> 'a` into the *same* function, but "lifted" to the option type, that is, having a signature `'a option -> 'a option -> 'a option` .

So, let's test it! Some test code might look like this:

```
// create some values
let p1 = Some 1
let p2 = Some 2
let zero = None

// test addition
addPositive p1 p2
addPositive p1 zero
addPositive zero p2
addPositive zero zero
```

You can see that unfortunately we do have to wrap the normal values in `Some` in order to get the `None` as identity.

That sounds tedious but in practice, it is easy enough. The code below shows how we might handle the two distinct cases when summing a list. First how to sum a non-empty list, and then how to sum an empty list.

```
[1..10]
|> List.map Some
|> List.fold addPositive zero

[]
|> List.map Some
|> List.fold addPositive zero
```

ValidationResult revisited

While we're at it, let's also revisit the `ValidationResult` type that we described earlier when talking about using lists to get closure. Here it is again:

```
type ValidationResult =
    | Success
    | Failure of string list
```

Now that we've got some insight into the positive integer case, let's look at this type from a different angle as well.

The type has two cases. One case holds data that we care about, and the other case holds no data. But the data we really care about are the error messages, not the success. As Leo Tolstoy nearly said "All validation successes are alike; each validation failure is a failure in its own way."

So, rather than thinking of it as a "Result", let's think of the type as *storing failures*, and rewrite it like this instead, with the failure case first:

```
type ValidationFailure =
    | Failure of string list
    | Success
```

Does this type appear familar now?

Yes! It's the option type again! Can we never get away from the darn thing?

Using the option type, we can simplify the design of the `ValidationFailure` type to just this:

```
type ValidationFailure = string list option
```

The helper to convert a string into the failure case is now just `Some` with a list:

```
let fail str =
    Some [str]
```

And the "add" function can reuse `optionAdd`, but this time with list concatenation as the underlying operation:

```
let addFailure f1 f2 = optionAdd (@) f1 f2
```

Finally, the "zero" that was the `Success` case in the original design now simply becomes `None` in the new design.

Here's all the code, plus tests

```
module MonoidalValidationOption =

    type ValidationFailure = string list option

    // helper to convert a string into the failure case
    let fail str =
        Some [str]

    let validateBadWord badWord (name:string) =
        if name.Contains(badWord) then
```

```
            fail ("string contains a bad word: " + badWord)
        else
            None

    let validateLength maxLength name =
        if String.length name > maxLength then
            fail "string is too long"
        else
            None

    let optionAdd f o1 o2 =
        match o1, o2 with
        | None, _ -> o2
        | _, None -> o1
        | Some s1, Some s2 -> Some (f s1 s2)

    /// add two results using optionAdd
    let addFailure f1 f2 = optionAdd (@) f1 f2

    // define the Zero
    let Success = None

module MonoidalValidationOptionTest =
    open MonoidalValidationOption

    let test1 =
        let result1 = Success
        let result2 = Success
        addFailure result1 result2
        |> printfn "Result is %A"

        // Result is <null>

    let test2 =
        let result1 = Success
        let result2 = fail "string is too long"
        addFailure result1 result2
        |> printfn "Result is %A"
        // Result is Some ["string is too long"]

    let test3 =
        let result1 = fail "string is null or empty"
        let result2 = fail "string is too long"
        addFailure result1 result2
        |> printfn "Result is %A"
        // Result is Some ["string is null or empty"; "string is too long"]

    let test4 =
        let validationResults str =
            [
            validateLength 10
            validateBadWord "monad"
            validateBadWord "cobol"
```

```
        ]
        |> List.map (fun validate -> validate str)

    "cobol has native support for monads"
    |> validationResults
    |> List.reduce addFailure
    |> printfn "Result is %A"
    // Result is Some
    //   ["string is too long"; "string contains a bad word: monad";
    //    "string contains a bad word: cobol"]

let test5 =
    let validationResults str =
        []
        |> List.map (fun validate -> validate str)

    "cobol has native support for monads"
    |> validationResults
    |> List.fold addFailure Success
    |> printfn "Result is %A"
    // Result is <null>
```

Summary of the design tips

Let's pause for a second and see what we have covered so far.

Here are all the design tips together:

- To easily create a monoidal type, make sure that each field of the type is also a monoid.
- To enable closure for a non-numeric type, replace single items with lists (or a similar data structure).
- To get associativity for an operation, try to move the operation into the object.
- To get identity for an operation, create a special case in a discriminated union, or, even simpler, just use Option.

In the next two sections, we'll apply these tips to two of the non-monoids that we have seen in previous posts: "average" and "most frequent word".

A case study: Average

So now we have the toolkit that will enable us to deal with the thorny case of averages.

Here's a simple implementation of a pairwise average function:

```
let avg i1 i2 =
    float (i1 + i2) / 2.0

// test
avg 4 5 |> printfn "Average is %g"
// Average is 4.5
```

As we mentioned briefly in the first post, `avg` fails on all three monoid requirements!

First, it is not closed. Two ints that are combined together using `avg` do not result in another int.

Second, even if it was closed, `avg` is not associative, as we can see by defining a similar float function `avgf` :

```
let avgf i1 i2 =
    (i1 + i2) / 2.0

// test
avgf (avgf 1.0 3.0) 5.0  |> printfn "Average from left is %g"
avgf 1.0 (avgf 3.0 5.0)  |> printfn "Average from right is %g"

// Average from left is 3.5
// Average from right is 2.5
```

Finally, there is no identity.

What number, when averaged with any other number, returns the original value? Answer: none!

Applying the design tips

So let's apply the design tips to see if they help us come up with a solution.

- *To easily create a monoidal type, make sure that each field of the type is also a monoid.*

Well, "average" is a mathematical operation, so we could expect that a monoidal equivalent would also be based on numbers.

- *To enable closure for a non-numeric type, replace single items with lists.*

This looks at first glance like it won't be relevant, so we'll skip this for now.

- *To get associativity for an operation, try to move the operation into the object.*

Here's the crux! How do we convert "average" from a verb (an operation) to a noun (a data structure)?

The answer is that we create a structure that is not actually an average, but a "delayed average" -- everything you need to make an average on demand.

That is, we need a data structure with *two* components: a total, and a count. With these two numbers we can calculate an average as needed.

```
// store all the info needed for an average
type Avg = {total:int; count:int}

// add two Avgs together
let addAvg avg1 avg2 =
    {total = avg1.total + avg2.total;
     count = avg1.count + avg2.count}
```

The good thing about this, is that structure stores `ints`, not `floats`, so we don't need to worry about loss of precision or associativity of floats.

The last tip is:

- *To get identity for an operation, create a special case in a discriminated union, or, even simpler, just use Option.*

In this case, the tip is not needed, as we can easily create a zero by setting the two components to be zero:

```
let zero = {total=0; count=0}
```

We could also have used `None` for the zero, but it seems like overkill in this case. If the list is empty, the `Avg` result is valid, even though we can't do the division.

Once we have had this insight into the data structure, the rest of the implementation follows easily. Here is all the code, plus some tests:

```
module Average =

    // store all the info needed for an average
    type Avg = {total:int; count:int}

    // add two Avgs together
    let addAvg avg1 avg2 =
        {total = avg1.total + avg2.total;
         count = avg1.count + avg2.count}

    // inline version of add
    let (++) = addAvg

    // construct an average from a single number
    let avg n = {total=n; count=1}

    // calculate the average from the data.
    // return 0 for empty lists
    let calcAvg avg =
        if avg.count = 0
        then 0.0
        else float avg.total / float avg.count

    // alternative - return None for empty lists
    let calcAvg2 avg =
        if avg.count = 0
        then None
        else Some (float avg.total / float avg.count)

    // the identity
    let zero = {total=0; count=0}

    // test
    addAvg (avg 4) (avg 5)
    |> calcAvg
    |> printfn "Average is %g"
    // Average is 4.5

    (avg 4) ++ (avg 5) ++ (avg 6)
    |> calcAvg
    |> printfn "Average is %g"
    // Average is 5

    // test
    [1..10]
    |> List.map avg
    |> List.reduce addAvg
    |> calcAvg
    |> printfn "Average is %g"
    // Average is 5.5
```

In the code above, you can see that I created a `calcAvg` function that uses the `Avg` structure to calculate a (floating point) average. One nice thing about this approach is that we can delay having to make a decision about what to do with a zero divisor. We can just return `0`, or alternatively `None`, or we can just postpone the calculation indefinitely, and only generate the average at the last possible moment, on demand!

And of course, this implementation of "average" has the ability to do incremental averages. We get this for free because it is a monoid.

That is, if I have already calculated the average of a million numbers, and I want to add one more, I don't have to recalculate everything, I can just add the new number to the totals so far.

A slight diversion on metrics

If you have ever been responsible for managing any servers or services, you will be aware of the importance of logging and monitoring metrics, such as CPU, I/O, etc.

One of the questions you often face then is how to design your metrics. Do you want kilobytes per second, or just total kilobytes since the server started. Visitors per hour, or total visitors?

If you look at some guidelines when creating metrics you will see the frequent recommendation to only track metrics that are *counters*, not *rates*.

The advantage of counters is that (a) missing data doesn't affect the big picture, and (b) they can be aggregated in many ways after the fact -- by minute, by hour, as a ratio with something else, and so on.

Now that you have worked through this series, you can see that the recommendation can really be rephrased as **metrics should be monoids**.

The work we did in the code above to transform "average" into two components, "total" and "count", is exactly what you want to do to make a good metric.

Averages and other rates are not monoids, but "total" and "count" are, and then "average" can be calculated from them at your leisure.

Case study: Turning "most frequent word" into a monoid homomorphism

In the last post, we implemented a "most frequent word" function, but found that it wasn't a monoid homomorphism. That is,

```
mostFrequentWord(text1) + mostFrequentWord(text2)
```

did *not* give the same result as:

```
mostFrequentWord( text1 + text2 )
```

Again, we can use the design tips to fix this up so that it works.

The insight here is again to delay the calculation until the last minute, just as we did in the "average" example.

Rather than calculating the most frequent word upfront then, we create a data structure that stores all the information that we need to calculate the most frequent word later.

```
module FrequentWordMonoid =

    open System
    open System.Text.RegularExpressions

    type Text = Text of string

    let addText (Text s1) (Text s2) =
        Text (s1 + s2)

    // return a word frequency map
    let wordFreq (Text s) =
        Regex.Matches(s,@"\S+")
        |> Seq.cast<Match>
        |> Seq.map (fun m -> m.ToString())
        |> Seq.groupBy id
        |> Seq.map (fun (k,v) -> k,Seq.length v)
        |> Map.ofSeq
```

In the code above we have a new function `wordFreq`, that returns a `Map<string,int>` rather just a single word. That is, we are now working with dictionaries, where each slot has a word and its associated frequency.

Here is a demonstration of how it works:

```
module FrequentWordMonoid =

    // code from above

    let page1() =
        List.replicate 1000 "hello world "
        |> List.reduce (+)
        |> Text

    let page2() =
        List.replicate 1000 "goodbye world "
        |> List.reduce (+)
        |> Text

    let page3() =
        List.replicate 1000 "foobar "
        |> List.reduce (+)
        |> Text

    let document() =
        [page1(); page2(); page3()]

    // show some word frequency maps
    page1() |> wordFreq |> printfn "The frequency map for page1 is %A"
    page2() |> wordFreq |> printfn "The frequency map for page2 is %A"

    //The frequency map for page1 is map [("hello", 1000); ("world", 1000)]
    //The frequency map for page2 is map [("goodbye", 1000); ("world", 1000)]

    document()
    |> List.reduce addText
    |> wordFreq
    |> printfn "The frequency map for the document is %A"

    //The frequency map for the document is map [
    //      ("foobar", 1000); ("goodbye", 1000);
    //      ("hello", 1000); ("world", 2000)]
```

With this map structure in place, we can create a function `addMap` to add two maps. It simply merges the frequency counts of the words from both maps.

```
module FrequentWordMonoid =

    // code from above

    // define addition for the maps
    let addMap map1 map2 =
        let increment mapSoFar word count =
            match mapSoFar |> Map.tryFind word with
            | Some count' -> mapSoFar |> Map.add word (count + count')
            | None -> mapSoFar |> Map.add word count

        map2 |> Map.fold increment map1
```

And when we have combined all the maps together, we can then calculate the most frequent word by looping through the map and finding the word with the largest frequency.

```
module FrequentWordMonoid =

    // code from above

    // as the last step,
    // get the most frequent word in a map
    let mostFrequentWord map =
        let max (candidateWord,maxCountSoFar) word count =
            if count > maxCountSoFar
            then (word,count)
            else (candidateWord,maxCountSoFar)

        map |> Map.fold max ("None",0)
```

So, here are the two scenarios revisited using the new approach.

The first scenario combines all the pages into a single text, then applies `wordFreq` to get a frequency map, and applies `mostFrequentWord` to get the most frequent word.

The second scenario applies `wordFreq` to each page separately to get a map for each page. These maps are then combined with `addMap` to get a single global map. Then `mostFrequentWord` is applied as the last step, as before.

```
module FrequentWordMonoid =

    // code from above

    document()
    |> List.reduce addText
    |> wordFreq
    // get the most frequent word from the big map
    |> mostFrequentWord
    |> printfn "Using add first, the most frequent word and count is %A"

    //Using add first, the most frequent word and count is ("world", 2000)

    document()
    |> List.map wordFreq
    |> List.reduce addMap
    // get the most frequent word from the merged smaller maps
    |> mostFrequentWord
    |> printfn "Using map reduce, the most frequent and count is %A"

    //Using map reduce, the most frequent and count is ("world", 2000)
```

If you run this code, you will see that you now get the *same* answer.

This means that `wordFreq` is indeed a monoid homomorphism, and is suitable for running in parallel, or incrementally.

Next time

We've seen a lot of code in this post, but it has all been focused on data structures.

However, there is nothing in the definition of a monoid that says that the things to be combined have to be data structures -- they could be *anything at all*.

In the next post we'll look at monoids applied to other objects, such as types, functions, and more.

Appendix: On Performance

In the examples above, I have made frequent use of `@` to "add" two lists in the same way that `+` adds two numbers. I did this to highlight the analogies with other monoidal operations such as numeric addition and string concatenation.

I hope that it is clear that the code samples above are meant to be teaching examples, not necessarily good models for the kind of real-world, battle-hardened, and all-too-ugly code you need in a production environment.

A couple of people have pointed out that using List append (`@`) should be avoided in general. This is because the entire first list needs to be copied, which is not very efficient.

By far the best way to add something to a list is to add it to the front using the so-called "cons" mechanism, which in F# is just `::` . F# lists are implemented as linked lists, so adding to the front is very cheap.

The problem with using this approach is that it is not symmetrical -- it doesn't add two lists together, just a list and an element. This means that it cannot be used as the "add" operation in a monoid.

If you don't need the benefits of a monoid, such as divide and conquer, then that is a perfectly valid design decision. No need to sacrifice performance for a pattern that you are not going to benefit from.

The other alternative to using `@` is to not use lists in the first place!

Alternatives to lists

In the `ValidationResult` design, I used a list to hold the error results so that we could get easy accumulation of the results. But I only chose the `list` type because it is really the default collection type in F#. I could have equally well have chosen sequences, or arrays, or sets. Almost any other collection type would have done the job just as well.

But not all types will have the same performance. For example, combining two sequences is a lazy operation. You don't have to copy all the data; you just enumerate one sequence, then the other. So that might be faster perhaps?

Rather than guessing, I wrote a little test script to measure performance at various list sizes, for various collection types.

I have chosen a very simple model: we have a list of objects, each of which is a collection containing *one* item. We then reduce this list of collections into a single giant collection using the appropriate monoid operation. Finally, we iterate over the giant collection once.

This is very similar to the `ValidationResult` design, where we would combine all the results into a single list of results, and then (presumably) iterate over them to show the errors.

It is also similar to the "most frequent word" design, above, where we combine all the individual frequency maps into a single frequency map, and then iterate over it to find the most frequent word. In that case, of course, we were using `map` rather than `list` , but the

set of steps is the same.

A performance experiment

Ok, here's the code:

```
module Performance =

    let printHeader() =
        printfn "Label,ListSize,ReduceAndIterMs"

    // time the reduce and iter steps for a given list size and print the results
    let time label reduce iter listSize =
        System.GC.Collect() //clean up before starting
        let stopwatch = System.Diagnostics.Stopwatch()
        stopwatch.Start()
        reduce() |> iter
        stopwatch.Stop()
        printfn "%s,%iK,%i" label (listSize/1000) stopwatch.ElapsedMilliseconds

    let testListPerformance listSize =
        let lists = List.init listSize (fun i -> [i.ToString()])
        let reduce() = lists |> List.reduce (@)
        let iter = List.iter ignore
        time "List.@" reduce iter listSize

    let testSeqPerformance_Append listSize =
        let seqs = List.init listSize (fun i -> seq {yield i.ToString()})
        let reduce() = seqs |> List.reduce Seq.append
        let iter = Seq.iter ignore
        time "Seq.append" reduce iter listSize

    let testSeqPerformance_Yield listSize =
        let seqs = List.init listSize (fun i -> seq {yield i.ToString()})
        let reduce() = seqs |> List.reduce (fun x y -> seq {yield! x; yield! y})
        let iter = Seq.iter ignore
        time "seq(yield!)" reduce iter listSize

    let testArrayPerformance listSize =
        let arrays = List.init listSize (fun i -> [| i.ToString() |])
        let reduce() = arrays |> List.reduce Array.append
        let iter = Array.iter ignore
        time "Array.append" reduce iter listSize

    let testResizeArrayPerformance listSize  =
        let resizeArrays = List.init listSize (fun i -> new ResizeArray<string>( [i.To
String()] ) )
        let append (x:ResizeArray<_>) y = x.AddRange(y); x
        let reduce() = resizeArrays |> List.reduce append
        let iter = Seq.iter ignore
        time "ResizeArray.append" reduce iter listSize
```

Let's go through the code quickly:

- The `time` function times the reduce and iteration steps. It deliberately does not test how long it takes to create the collection. I do perform a GC before starting, but in reality, the memory pressure that a particular type or algorithm causes is an important

part of the decision to use it (or not). Understanding how GC works is an important part of getting performant code.

- The `testListPerformance` function sets up the list of collections (lists in this case) and also the `reduce` and `iter` functions. It then runs the timer on `reduce` and `iter`.
- The other functions do the same thing, but with sequences, arrays, and ResizeArrays (standard .NET Lists). Out of curiosity, I thought I'd test two ways of merging sequences, one using the standard library function `Seq.append` and the other using two `yield!`s in a row.
- The `testResizeArrayPerformance` uses ResizeArrays and adds the right list to the left one. The left one mutates and grows larger as needed, using a growth strategy that keeps inserts efficient.

Now let's write code to check the performance on various sized lists. I chose to start with a count of 2000 and move by increments of 4000 up to 50000.

```
open Performance

printHeader()

[2000..4000..50000]
|> List.iter testArrayPerformance

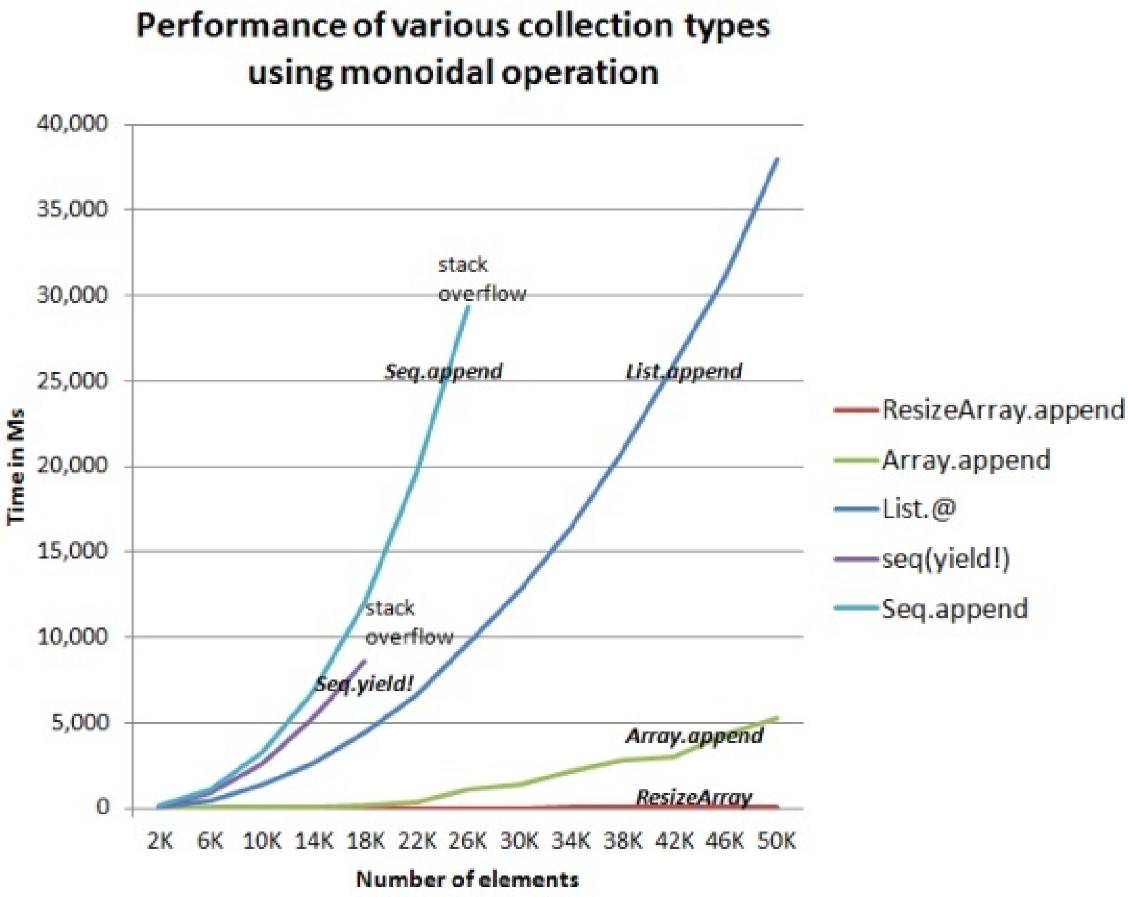
[2000..4000..50000]
|> List.iter testResizeArrayPerformance

[2000..4000..50000]
|> List.iter testListPerformance

[2000..4000..50000]
|> List.iter testSeqPerformance_Append

[2000..4000..50000]
|> List.iter testSeqPerformance_Yield
```

I won't list all the detailed output -- you can run the code for yourself -- but here is a chart of the results.



There are a few things to note:

- The two sequence-based examples crashed with stack overflows. The `yield!` was about 30% faster than `Seq.append`, but also ran out of stack faster.
- List.append didn't run out of stack, but got much slower as the lists got larger.
- Array.append was fast, and increases more slowly with the size of the list
- ResizeArray was fastest of all, and didn't break a sweat even with large lists.

For the three collection types that didn't crash, I also timed them for a list of 100K items. The results were:

- List = 150,730 ms
- Array = 26,062 ms
- ResizeArray = 33 ms

A clear winner there, then.

Analyzing the results

What conclusion can we draw from this little experiment?

First, you might have all sorts of questions, such as: Were you running in debug or release mode? Did you have optimization turned on? What about using parallelism to increase performance? And no doubt, there will be comments saying "why did you use technique X, technique Y is so much better".

But here's the conclusion I would like to make:

- **You cannot draw any conclusion from these results!**

Every situation is different and requires a different approach:

- If you are working with small data sets you might not care about performance anyway. In this case I would stick with lists -- I'd rather not sacrifice pattern matching and immutability unless I have to.
- The performance bottleneck might not be in the list addition code. There is no point working on optimizing the list addition if you are actually spending all your time on disk I/O or network delays. A real-world version of the word frequency example might actually spend most of its time doing reading from disk, or parsing, rather than adding lists.
- If you working at the scale of Google, Twitter, or Facebook, you really need to go and hire some algorithm experts.

The only principles that we can take away from any discussion on optimization and performance are:

- **A problem must be dealt with in its own context.** The size of the data being processed, the type of hardware, the amount of memory, and so on. All these will make a difference to your performance. What works for me may not work for you, which is why...
- **You should always measure, not guess.** Don't make assumptions about where your code is spending its time -- learn to use a profiler! There are some good examples of using a profiler here and here.
- **Be wary of micro-optimizations**. Even if your profiler shows that your sorting routine spends all its time in comparing strings, that doesn't necessarily mean that you need to improve your string comparison function. You might be better off improving your algorithm so that you don't need to do so many comparisons in the first place. Premature optimization and all that.

In this series, we'll look at how so-called "applicative parsers" work. In order to understand something, there's nothing like building it for yourself, and so we'll create a basic parser library from scratch, then some useful "parser combinators", and then finish off by building a complete JSON parser.

- Understanding Parser Combinators. Building a parser combinator library from scratch.
- Building a useful set of parser combinators. 15 or so combinators that can be combined to parse almost anything.
- Improving the parser library. Adding more informative errors.
- Writing a JSON parser from scratch. In 250 lines of code.

Understanding Parser Combinators

UPDATE: Slides and video from my talk on this topic

In this series, we'll look at how so-called "applicative parsers" work. In order to understand something, there's nothing like building it for yourself, and so we'll create a basic parser library from scratch, and then some useful "parser combinators", and then finish off by building a complete JSON parser.

Now terms like "applicative parsers" and "parser combinators" can make this approach seem complicated, but rather than attempting to explain these concepts up front, we'll just dive in and start coding.

We'll build up to the complex stuff incrementally via a series of implementations, where each implementation is only slightly different from the previous one. By using this approach, I hope that at each stage the design and concepts will be easy to understand, and so by the end of this series, parser combinators will have become completely demystified.

There will be four posts in this series:

- In this, the first post, we'll look at the basic concepts of parser combinators and build the core of the library.
- In the second post, we'll build up a useful library of combinators.
- In the third post, we'll work on providing helpful error messages.
- In the last post, we'll build a JSON parser using this parser library.

Obviously, the focus here will not be on performance or efficiency, but I hope that it will give you the understanding that will then enable you to use libraries like FParsec effectively. And by the way, a big thank you to Stephan Tolksdorf, who created FParsec. You should make it your first port of call for all your .NET parsing needs!

Implementation 1. Parsing a hard-coded character

For our first implementation, let's create something that just parses a single, hard-coded, character, in this case, the letter "A". You can't get much simpler than that!

Here is how it works:

- The input to a parser is a stream of characters. We could use something complicated,

but for now we'll just use a `string`.
- If the stream is empty, then return a pair consisting of `false` and an empty string.
- If the first character in the stream is an `A`, then return a pair consisting of `true` and the remaining stream of characters.
- If the first character in the stream is not an `A`, then return `false` and the (unchanged) original stream of characters.

Here's the code:

```
let A_Parser str =
    if String.IsNullOrEmpty(str) then
        (false,"")
    else if str.[0] = 'A' then
        let remaining = str.[1..]
        (true,remaining)
    else
        (false,str)
```

The signature of `A_Parser` is:

```
val A_Parser :
    string -> (bool * string)
```

which tells us that the input is a string, and the output is a pair consisting of the boolean result and another string (the remaining input), like this:



Let's test it now -- first with good input:

```
let inputABC = "ABC"
A_Parser inputABC
```

The result is:

```
(true, "BC")
```

As you can see, the `A` has been consumed and the remaining input is just `"BC"`.

And now with bad input:

```
let inputZBC = "ZBC"
A_Parser inputZBC
```

which gives the result:

```
(false, "ZBC")
```

And in this case, the first character was *not* consumed and the remaining input is still `"ZBC"`.

So, there's an incredibly simple parser for you. If you understand that, then everything that follows will be easy!

Implementation 2. Parsing a specified character

Let's refactor so that we can pass in the character we want to match, rather than having it be hard coded.

And this time, rather than returning true or false, we'll return a message indicating what happened.

We'll call the function `pchar` for "parse char". Here's the code:

```
let pchar (charToMatch,str) =
    if String.IsNullOrEmpty(str) then
        let msg = "No more input"
        (msg,"")
    else
        let first = str.[0]
        if first = charToMatch then
            let remaining = str.[1..]
            let msg = sprintf "Found %c" charToMatch
            (msg,remaining)
        else
            let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch first
            (msg,str)
```

This code is just like the previous example, except that the unexpected character is now shown in the error message.

The signature of `pchar` is:

```
val pchar :
    (char * string) -> (string * string)
```

which tells us that the input is a pair of (string,character to match) and the output is a pair consisting of the (string) result and another string (the remaining input).

Let's test it now -- first with good input:

```
let inputABC = "ABC"
pchar('A',inputABC)
```

The result is:

```
("Found A", "BC")
```

As before, the `A` has been consumed and the remaining input is just `"BC"`.

And now with bad input:

```
let inputZBC = "ZBC"
pchar('A',inputZBC)
```

which gives the result:

```
("Expecting 'A'. Got 'Z'", "ZBC")
```

And again, as before, the first character was *not* consumed and the remaining input is still `"ZBC"`.

If we pass in `Z`, then the parser does succeed:

```
pchar('Z',inputZBC)  // ("Found Z", "BC")
```

Implementation 3. Returning a Success/Failure

We want to be able to tell the difference between a successful match and a failure, and returning a stringly-typed message is not very helpful, so let's define a special "choice" type to indicate the difference. I'll call it `Result`:

```
type Result<'a> =
    | Success of 'a
    | Failure of string
```

The `Success` case is generic and can contain any value. The `Failure` case contains an error message.

Note: for more on using this Success/Failure approach, see my talk on functional error handling.

We can now rewrite the parser to return one of the `Result` cases, like this:

```
let pchar (charToMatch,str) =
    if String.IsNullOrEmpty(str) then
        Failure "No more input"
    else
        let first = str.[0]
        if first = charToMatch then
            let remaining = str.[1..]
            Success (charToMatch,remaining)
        else
            let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch first
            Failure msg
```

The signature of `pchar` is now:

```
val pchar :
    (char * string) -> Result<char * string>
```

which tells us that the the output is now a `Result` (which in the `Success` case, contains the matched char and the remaining input string).

Let's test it again -- first with good input:

```
let inputABC = "ABC"
pchar('A',inputABC)
```

The result is:

```
Success ('A', "BC")
```

As before, the `A` has been consumed and the remaining input is just `"BC"`. We also get the *actual* matched char (`A` in this case).

And now with bad input:

```
let inputZBC = "ZBC"
pchar('A',inputZBC)
```

which gives the result:

```
Failure "Expecting 'A'. Got 'Z'"
```

And in this case, the `Failure` case is returned with the appropriate error message.

This is a diagram of the function's inputs and outputs now:



Implementation 4. Switching to a curried implementation

In the previous implementation, the input to the function has been a tuple -- a pair. This requires you to pass both inputs at once.

In functional languages like F#, it's more idiomatic to use a curried version, like this:

```
let pchar charToMatch str =
    if String.IsNullOrEmpty(str) then
        Failure "No more input"
    else
        let first = str.[0]
        if first = charToMatch then
            let remaining = str.[1..]
            Success (charToMatch,remaining)
        else
            let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch first
            Failure msg
```

Can you see the difference? The only difference is in the first line, and even then it is subtle.

Here's the uncurried (tuple) version:

```
let pchar (charToMatch,str) =
    ...
```

And here's the curried version:

```
let pchar charToMatch str =
    ...
```

The difference is much more obvious when you look at the type signatures. Here's the signature for the uncurried (tuple) version:

```
val pchar :
    (char * string) -> Result<char * string>
```

And here's the signature for the curried version:

```
val pchar :
    char -> string -> Result<char * string>
```

Here is the curried version of `pchar` represented as a diagram:



What is currying?

If you are unclear on how currying works, I have a post about it here, but basically it means that a multi-parameter function can be written as a series of one-parameter functions.

In other words, this two-parameter function:

```
let add x y =
    x + y
```

can be written as a one-parameter function that returns a lambda, like this:

```
let add x =
    fun y -> x + y  // return a lambda
```

or as a function that returns an inner function, like this:

```
let add x =
    let innerFn y = x + y
    innerFn // return innerFn
```

Rewriting with an inner function

We can take advantage of currying and rewrite the parser as a one-parameter function (where the parameter is `charToMatch`) that returns a inner function.

Here's the new implementation, with the inner function cleverly named `innerFn`:

```
let pchar charToMatch =
    // define a nested inner function
    let innerFn str =
        if String.IsNullOrEmpty(str) then
            Failure "No more input"
        else
            let first = str.[0]
            if first = charToMatch then
                let remaining = str.[1..]
                Success (charToMatch,remaining)
            else
                let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch first
                Failure msg
    // return the inner function
    innerFn
```

The type signature for this implementation looks like this:

```
val pchar :
    char -> string -> Result<char * string>
```

It's *exactly the same* as the previous version!

That is, both of the above implementations are identical in practice:

```
// two-parameter implementation
let pchar charToMatch str =
    ...

// one-parameter implementation with inner function
let pchar charToMatch =
    let innerFn str =
        ...
    // return the inner function
    innerFn
```

The benefits of the curried implementation

What's nice about the curried implementation is that we can partially apply the character we want to parse, like this:

```
let parseA = pchar 'A'
```

and then later on supply the second "input stream" parameter:

```
let inputABC = "ABC"
parseA inputABC  // Success ('A', "BC")

let inputZBC = "ZBC"
parseA inputZBC  // Failure "Expecting 'A'. Got 'Z'"
```

At this point, let's stop and review what is going on:

- The `pchar` function has two inputs
- We can provide one input (the char to match) and this results in a *function* being returned.
- We can then provide the second input (the stream of characters) to this parsing function, and this creates the final `Result` value.

Here's a diagram of `pchar` again, but this time with the emphasis on partial application:



It's very important that you understand this logic before moving on, because the rest of the post will build on this basic design.

Implementation 5. Encapsulating the parsing function in a type

If we look at `parseA` (from the example above) we can see that it has a function type:

```
val parseA : string -> Result<char * string>
```

That type is a bit complicated to use, so let's encapsulate it in a "wrapper" type called `Parser`, like this:

```
type Parser<'T> = Parser of (string -> Result<'T * string>)
```

By encapsulating it, we'll go from this design:



to this design:



The change to the implementation is very simple. We just need to change the way the inner function is returned.

That is, from this:

```
let pchar charToMatch =
    let innerFn str =
        ...
    // return the inner function
    innerFn
```

to this:

```
let pchar charToMatch =
    let innerFn str =
        ...
    // return the "wrapped" inner function
    Parser innerFn
```

Testing the wrapped function

Ok, now let's test again:

```
let parseA = pchar 'A'
let inputABC = "ABC"
parseA inputABC  // compiler error
```

But now we get a compiler error:

```
error FS0003: This value is not a function and cannot be applied
```

And of course that is because the function is wrapped in the `Parser` data structure! It's not longer directly accessible.

So now we need a helper function that can extract the inner function and run it against the input stream. Let's call it `run` !

Here's the implementation of `run` :

```
let run parser input =
    // unwrap parser to get inner function
    let (Parser innerFn) = parser
    // call inner function with input
    innerFn input
```

And now we can run the `parseA` parser against various inputs again:

```
let inputABC = "ABC"
run parseA inputABC  // Success ('A', "BC")

let inputZBC = "ZBC"
run parseA inputZBC  // Failure "Expecting 'A'. Got 'Z'"
```

That's it! We've got a basic `Parser` type! I hope that this all makes sense so far.

Combining two parsers in sequence: the "and then" combinator

That last implementation is good enough for basic parsing logic. We'll revisit it later, but now let's move up a level and develop some ways of combining parsers together -- the "parser combinators" mentioned at the beginning.

We'll start with combining two parsers in sequence. For example, say that we want a parser that matches "A" and then "B". We could try writing something like this:

```
let parseA = pchar 'A'
let parseB = pchar 'B'

let parseAThenB = parseA >> parseB
```

but that gives us a compiler error, as the output of `parseA` does not match the input of `parseB`, and so they cannot be composed like that.

If you are familiar with functional programming patterns, the need to chain a sequence of wrapped types together like this happens frequently, and the solution is a `bind` function.

However, in this case, I won't implement `bind` but will instead go straight to an `andThen` implemention.

The implementation logic will be as follows:

- Run the first parser.
- If there is a failure, return.
- Otherwise, run the second parser with the remaining input.
- If there is a failure, return.
- If both parsers succeed, return a pair (tuple) that contains both parsed values.

Here's the code for `andThen`:

```
let andThen parser1 parser2 =
    let innerFn input =
        // run parser1 with the input
        let result1 = run parser1 input

        // test the result for Failure/Success
        match result1 with
        | Failure err ->
            // return error from parser1
            Failure err

        | Success (value1,remaining1) ->
            // run parser2 with the remaining input
            let result2 =  run parser2 remaining1

            // test the result for Failure/Success
            match result2 with
            | Failure err ->
                // return error from parser2
                Failure err

            | Success (value2,remaining2) ->
                // combine both values as a pair
                let newValue = (value1,value2)
                // return remaining input after parser2
                Success (newValue,remaining2)

    // return the inner function
    Parser innerFn
```

The implementation follows the logic described above.

We'll also define an infix version of `andThen` so that we can use it like regular `>>` composition:

```
let ( .>>. ) = andThen
```

Note: the parentheses are needed to define a custom operator, but are not needed in the infix usage.

If we look at the signature of `andThen` :

```
val andThen :
     parser1:Parser<'a> -> parser2:Parser<'b> -> Parser<'a * 'b>
```

we can see that it works for any two parsers, and they can be of different types (`'a` and `'b`).

Testing `andThen`

Let's test it and see if it works!

First, create the compound parser:

```
let parseA = pchar 'A'
let parseB = pchar 'B'
let parseAThenB = parseA .>>. parseB
```

If you look at the types, you can see that all three values have type `Parser` :

```
val parseA : Parser<char>
val parseB : Parser<char>
val parseAThenB : Parser<char * char>
```

`parseAThenB` is of type `Parser<char * char>` meaning that the parsed value is a pair of chars.

Now since the combined parser `parseAThenB` is just another `Parser` , we can use `run` with it as before.

```
run parseAThenB "ABC"  // Success (('A', 'B'), "C")

run parseAThenB "ZBC"  // Failure "Expecting 'A'. Got 'Z'"

run parseAThenB "AZC"  // Failure "Expecting 'B'. Got 'Z'"
```

You can see that in the success case, the pair `('A', 'B')` was returned, and also that failure happens when either letter is missing from the input.

Choosing between two parsers: the "or else" combinator

Let's look at another important way of combining parsers -- the "or else" combinator.

For example, say that we want a parser that matches "A" *or* "B". How could we combine them?

The implementation logic would be:

- Run the first parser.

- On success, return the parsed value, along with the remaining input.
- Otherwise, on failure, run the second parser with the original input...
- ...and in this case, return the result (success or failure) from the second parser.

Here's the code for `orElse` :

```
let orElse parser1 parser2 =
    let innerFn input =
        // run parser1 with the input
        let result1 = run parser1 input

        // test the result for Failure/Success
        match result1 with
        | Success result ->
            // if success, return the original result
            result1

        | Failure err ->
            // if failed, run parser2 with the input
            let result2 = run parser2 input

            // return parser2's result
            result2

    // return the inner function
    Parser innerFn
```

And we'll define an infix version of `orElse` as well:

```
let ( <|> ) = orElse
```

If we look at the signature of `orElse` :

```
val orElse :
    parser1:Parser<'a> -> parser2:Parser<'a> -> Parser<'a>
```

we can see that it works for any two parsers, but they must both be the *same* type `'a` .

Testing `orElse`

Time to test it. First, create the combined parser:

```
let parseA = pchar 'A'
let parseB = pchar 'B'
let parseAOrElseB = parseA <|> parseB
```

If you look at the types, you can see that all three values have type `Parser<char>` :

```
val parseA : Parser<char>
val parseB : Parser<char>
val parseAOrElseB : Parser<char>
```

Now if we run `parseAOrElseB` we can see that it successfully handles an "A" or a "B" as first character.

```
run parseAOrElseB "AZZ"  // Success ('A', "ZZ")

run parseAOrElseB "BZZ"  // Success ('B', "ZZ")

run parseAOrElseB "CZZ"  // Failure "Expecting 'B'. Got 'C'"
```

Combining `andThen` and `orElse`

With these two basic combinators, we can build more complex ones, such as "A and then (B or C)".

Here's how to build up `aAndThenBorC` from simpler parsers:

```
let parseA = pchar 'A'
let parseB = pchar 'B'
let parseC = pchar 'C'
let bOrElseC = parseB <|> parseC
let aAndThenBorC = parseA .>>. bOrElseC
```

And here it is in action:

```
run aAndThenBorC "ABZ"  // Success (('A', 'B'), "Z")
run aAndThenBorC "ACZ"  // Success (('A', 'C'), "Z")
run aAndThenBorC "QBZ"  // Failure "Expecting 'A'. Got 'Q'"
run aAndThenBorC "AQZ"  // Failure "Expecting 'C'. Got 'Q'"
```

Note that the last example gives a misleading error. It says "Expecting 'C'" when it really should say "Expecting 'B' or 'C'". We won't attempt to fix this right now, but in a later post we'll implement better error messages.

Choosing from a list of parsers: "choice" and "anyOf"

This is where where the power of combinators starts kicking in, because with `orElse` in our toolbox, we can use it to build even more combinators.

For example, let's say that we want choose from a *list* of parsers, rather than just two.

Well, that's easy. If we have a pairwise way of combining things, we can extend that to combining an entire list using `reduce` (for more on working with `reduce`, see this post on monoids).

```
/// Choose any of a list of parsers
let choice listOfParsers =
    List.reduce ( <|> ) listOfParsers
```

Note that this will fail if the input list is empty, but we will ignore that for now.

The signature of `choice` is:

```
val choice :
    Parser<'a> list -> Parser<'a>
```

which shows us that, as expected, the input is a list of parsers, and the output is a single parser.

With `choice` available, we can create an `anyOf` parser that matches any character in a list, using the following logic:

- The input is a list of characters
- Each char in the list is transformed into a parser for that char using `pchar`
- Finally, all the parsers are combined using `choice`

Here's the code:

```
/// Choose any of a list of characters
let anyOf listOfChars =
    listOfChars
    |> List.map pchar // convert into parsers
    |> choice
```

Let's test it by creating a parser for any lowercase character and any digit character:

```
let parseLowercase =
    anyOf ['a'..'z']

let parseDigit =
    anyOf ['0'..'9']
```

If we test them, they work as expected:

```
run parseLowercase "aBC"  // Success ('a', "BC")
run parseLowercase "ABC"  // Failure "Expecting 'z'. Got 'A'"

run parseDigit "1ABC"  // Success ("1", "ABC")
run parseDigit "9ABC"  // Success ("9", "ABC")
run parseDigit "|ABC"  // Failure "Expecting '9'. Got '|'"
```

Again, the error messages are misleading. Any lowercase letter can be expected, not just 'z', and any digit can be expected, not just '9'. As I said earlier, we'll work on the error messages in a later post.

Review

Let's stop for now, and review what we have done:

- We have created a type `Parser` that is a wrapper for a parsing function.
- The parsing function takes an input (e.g. string) and attempts to match the input using the criteria baked into the function.
- If the match succeeds, the parsing function returns a `Success` with the matched item and the remaining input.
- If the match fails, the parsing function returns a `Failure` with reason for the failure.
- And finally, we saw some "combinators" -- ways in which `Parser` s could be combined to make a new `Parser` : `andThen` and `orElse` and `choice` .

Listing of the parser library so far

Here's the complete listing for the parsing library so far -- it's about 90 lines of code.

The source code displayed below is also available at this gist.

```
open System

/// Type that represents Success/Failure in parsing
type Result<'a> =
    | Success of 'a
    | Failure of string

/// Type that wraps a parsing function
type Parser<'T> = Parser of (string -> Result<'T * string>)

/// Parse a single character
let pchar charToMatch =
```

```
    // define a nested inner function
    let innerFn str =
        if String.IsNullOrEmpty(str) then
            Failure "No more input"
        else
            let first = str.[0]
            if first = charToMatch then
                let remaining = str.[1..]
                Success (charToMatch,remaining)
            else
                let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch first
                Failure msg
    // return the "wrapped" inner function
    Parser innerFn

/// Run a parser with some input
let run parser input =
    // unwrap parser to get inner function
    let (Parser innerFn) = parser
    // call inner function with input
    innerFn input

/// Combine two parsers as "A andThen B"
let andThen parser1 parser2 =
    let innerFn input =
        // run parser1 with the input
        let result1 = run parser1 input

        // test the result for Failure/Success
        match result1 with
        | Failure err ->
            // return error from parser1
            Failure err

        | Success (value1,remaining1) ->
            // run parser2 with the remaining input
            let result2 =  run parser2 remaining1

            // test the result for Failure/Success
            match result2 with
            | Failure err ->
                // return error from parser2
                Failure err

            | Success (value2,remaining2) ->
                // combine both values as a pair
                let newValue = (value1,value2)
                // return remaining input after parser2
                Success (newValue,remaining2)

    // return the inner function
    Parser innerFn
```

```
/// Infix version of andThen
let ( .>>. ) = andThen

/// Combine two parsers as "A orElse B"
let orElse parser1 parser2 =
    let innerFn input =
        // run parser1 with the input
        let result1 = run parser1 input

        // test the result for Failure/Success
        match result1 with
        | Success result ->
            // if success, return the original result
            result1

        | Failure err ->
            // if failed, run parser2 with the input
            let result2 = run parser2 input

            // return parser2's result
            result2

    // return the inner function
    Parser innerFn

/// Infix version of orElse
let ( <|> ) = orElse

/// Choose any of a list of parsers
let choice listOfParsers =
    List.reduce ( <|> ) listOfParsers

/// Choose any of a list of characters
let anyOf listOfChars =
    listOfChars
    |> List.map pchar // convert into parsers
    |> choice
```

Summary

In this post, we have created the foundations of a parsing library, and few simple combinators.

In the next post, we'll build on this to create a library with many more combinators.

The source code for this post is available at this gist.

Further information

- If you are interesting in using this technique in production, be sure to investigate the FParsec library for F#, which is optimized for real-world usage.
- For more information about parser combinators in general, search the internet for "Parsec", the Haskell library that influenced FParsec (and this post).
- For some examples of using FParsec, try one of these posts:
  - Implementing a phrase search query for FogCreek's Kiln
  - A LOGO Parser
  - A Small Basic Parser
  - A C# Parser and building a C# compiler in F#
  - Write Yourself a Scheme in 48 Hours in F#
  - Parsing GLSL, the shading language of OpenGL

Building a useful set of parser combinators

UPDATE: Slides and video from my talk on this topic

In this series, we are looking at how applicative parsers and parser combinators work.

- In the first post, we created the foundations of a parsing library.
- In this post, we'll build out the library with many other useful combinators. The combinator names will be copied from those used by FParsec, so that you can easily migrate to it.

---

1. `map` -- transforming the contents of a parser

When parsing, we often we want to match a particular string, such as a reserved word like "if" or "where". A string is just a sequence of characters, so surely we could use the same technique that we used to define `anyOf` in the first post, but using `andThen` instead of `orElse` ?

Here's a (failed) attempt to create a `pstring` parser using that approach:

```
let pstring str =
    str
    |> Seq.map pchar // convert into parsers
    |> Seq.reduce andThen
```

This doesn't work, because the output of `andThen` is different from the input (a tuple, not a char) and so the `reduce` approach fails.

In order to solve this, we'll need to use a different technique.

To get started, let's try just matching a string of a specific length. Say, for example, that we want to match a three digits in a row. Well, we can do that using `andThen` :

```
let parseDigit =
    anyOf ['0'..'9']

let parseThreeDigits =
    parseDigit .>>. parseDigit .>>. parseDigit
```

If we run it like this:

```
run parseThreeDigits "123A"
```

then we get the result:

```
Success ((('1', '2'), '3'), "A")
```

It does work, but the result contains a tuple inside a tuple `(('1', '2'), '3')` which is fugly and hard to use. It would be so much more convenient to just have a simple string (`"123"`).

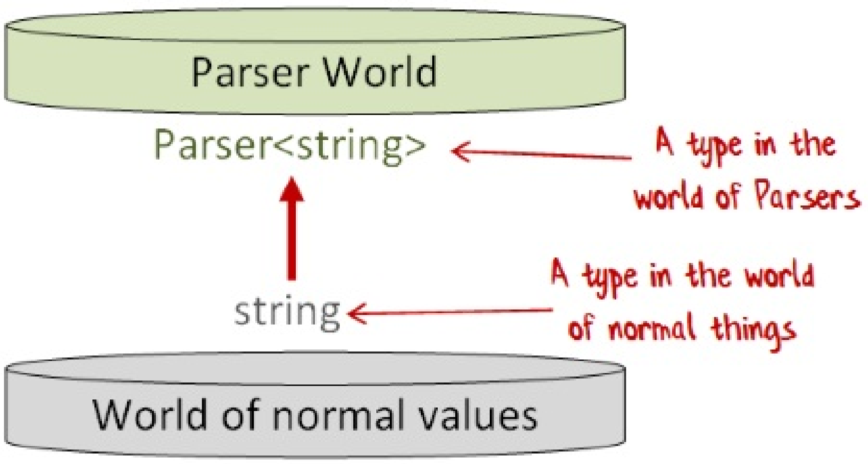
But in order to turn `('1', '2'), '3')` into `"123"` , we'll need a function that can reach inside of the parser and transform the result using an arbitrary passed in function.

Of course, what we need is the functional programmer's best friend, `map` .

To understand `map` and similar functions, I like to think of there being two worlds: a "Normal World", where regular things live, and "Parser World", where `Parser` s live.

You can think of Parser World as a sort of "mirror" of Normal World because it obeys the following rules:

- Every type in Normal World (say `char`) has a corresponding type in Parser World (`Parser<char>`).

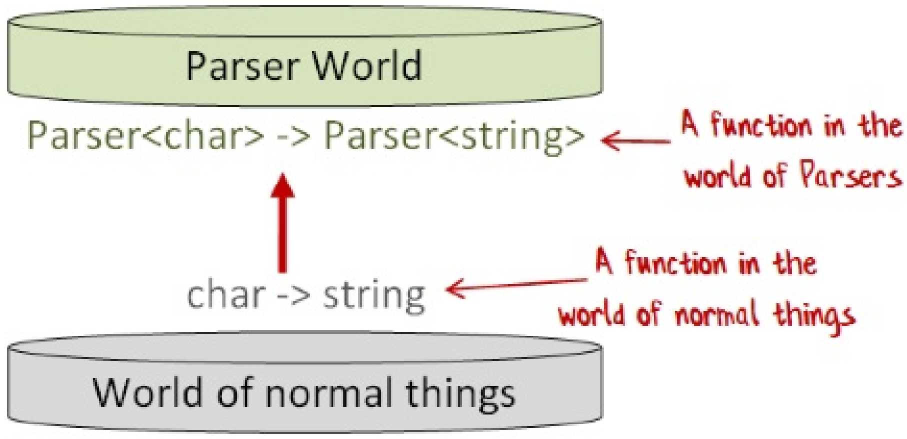


And:

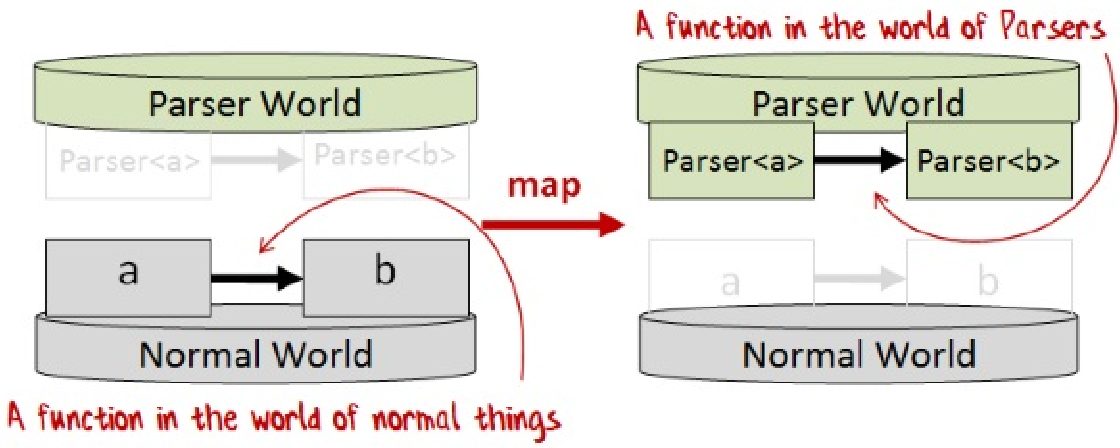
- Every value in Normal World (say `"ABC"`) has a corresponding value in Parser World (that is, some `Parser<string>` that returns `"ABC"`).

And:

- Every function in Normal World (say `char -> string`) has a corresponding function in Parser World (`Parser<char> -> Parser<string>`).



Using this metaphor then, `map` transforms (or "lifts") a function in Normal World into a function in Parser World.



And by the way, if you like this metaphor, I have a [whole series of posts that develop it further](#).

So that's what `map` does; how do we implement it?

The logic is:

- Inside the `innerFn`, run the parser to get the result.
- If the result was a success, apply the specified function to the success value to get a new, transformed value, and...
- ...return the new, mapped, value instead of the original value.

Here's the code (I've named the map function `mapP` to avoid confusion with other map functions):

```
let mapP f parser =
    let innerFn input =
        // run parser with the input
        let result = run parser input

        // test the result for Failure/Success
        match result with
        | Success (value,remaining) ->
            // if success, return the value transformed by f
            let newValue = f value
            Success (newValue, remaining)

        | Failure err ->
            // if failed, return the error
            Failure err
    // return the inner function
    Parser innerFn
```

If we look at the signature of `mapP` :

```
val mapP :
    f:('a -> 'b) -> Parser<'a> -> Parser<'b>
```

we can see that it has exactly the signature we want, transforming a function `'a -> 'b` into a function `Parser<'a> -> Parser<'b>` .

It's common to define an infix version of `map` as well:

```
let ( <!> ) = mapP
```

And in the context of parsing, we'll often want to put the mapping function *after* the parser, with the parameters flipped. This makes using `map` with the pipeline idiom much more convenient:

```
let ( |>> ) x f = mapP f x
```

Parsing three digits with `mapP`

With `mapP` available, we can revisit `parseThreeDigits` and turn the tuple into a string.

Here's the code:

```
let parseDigit = anyOf ['0'..'9']

let parseThreeDigitsAsStr =
    // create a parser that returns a tuple
    let tupleParser =
        parseDigit .>>. parseDigit .>>. parseDigit

    // create a function that turns the tuple into a string
    let transformTuple ((c1, c2), c3) =
        String [| c1; c2; c3 |]

    // use "map" to combine them
    mapP transformTuple tupleParser
```

Or, if you prefer a more compact implementation:

```
let parseThreeDigitsAsStr =
    (parseDigit .>>. parseDigit .>>. parseDigit)
    |>> fun ((c1, c2), c3) -> String [| c1; c2; c3 |]
```

And if we test it, we get a string in the result now, rather than a tuple:

```
run parseThreeDigitsAsStr "123A"  // Success ("123", "A")
```

We can go further, and map the string into an int:

```
let parseThreeDigitsAsInt =
    mapP int parseThreeDigitsAsStr
```

If we test this, we get an `int` in the Success branch.

```
run parseThreeDigitsAsInt "123A"  // Success (123, "A")
```

Let's check the type of `parseThreeDigitsAsInt` :

```
val parseThreeDigitsAsInt : Parser<int>
```

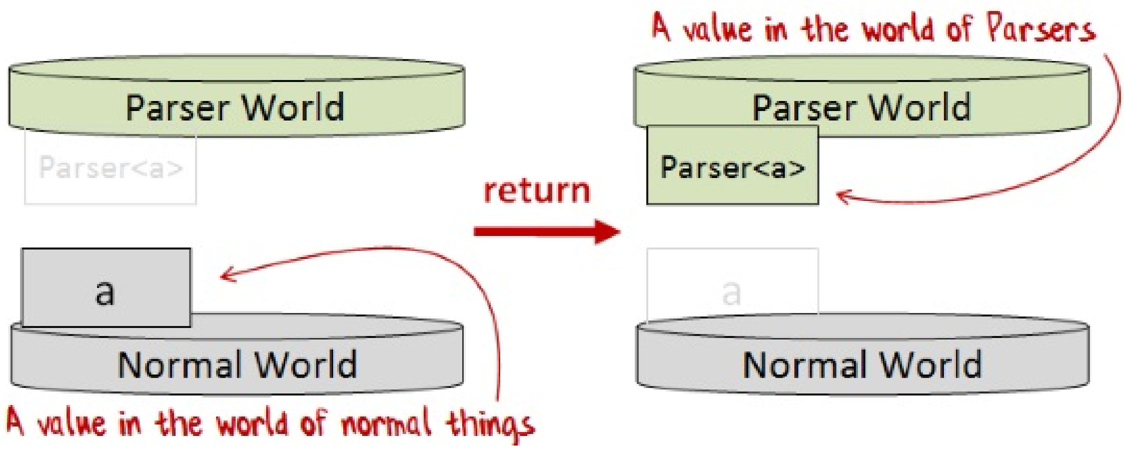
It's a `Parser<int>` now, not a `Parser<char>` or `Parser<string>` . The fact that a `Parser` can contain *any* type, not just a char or string, is a key feature that will be very valuable when we need to build more complex parsers.

2. `apply` and `return` -- lifting functions to the world of Parsers

To achieve our goal of creating a parser that matches a list of characters, we need two more helper functions which I will call `returnP` and `applyP`.

- `returnP` simply transforms a normal value into a value in Parser World
- `applyP` transforms a Parser containing a function (`Parser< 'a->'b >`) into a function in Parser World (`Parser<'a> -> Parser<'b >`)

Here's a diagram of `returnP`:



And here is the implementation of `returnP`:

```
let returnP x =
    let innerFn input =
        // ignore the input and return x
        Success (x,input )
    // return the inner function
    Parser innerFn
```

The signature of `returnP` is just as we want:

```
val returnP :
    'a -> Parser<'a>
```

Now here's a diagram of `applyP`:



And here is the implementation of `applyP`, which uses `.>>.` and `map`:

```
let applyP fP xP =
    // create a Parser containing a pair (f,x)
    (fP .>>. xP)
    // map the pair by applying f to x
    |> mapP (fun (f,x) -> f x)
```

The infix version of `applyP` is written as `<*>`:

```
let ( <*> ) = applyP
```

Again, the signature of `applyP` is just as we want:

```
val applyP :
    Parser<('a -> 'b)> -> Parser<'a> -> Parser<'b>
```

Why do we need these two functions? Well, `map` will lift functions in Normal World into functions in Parser World, but only for one-parameter functions.

What's great about `returnP` and `applyP` is that, together, they can lift *any* function in Normal World into a function in Parser World, no matter how many parameters it has.

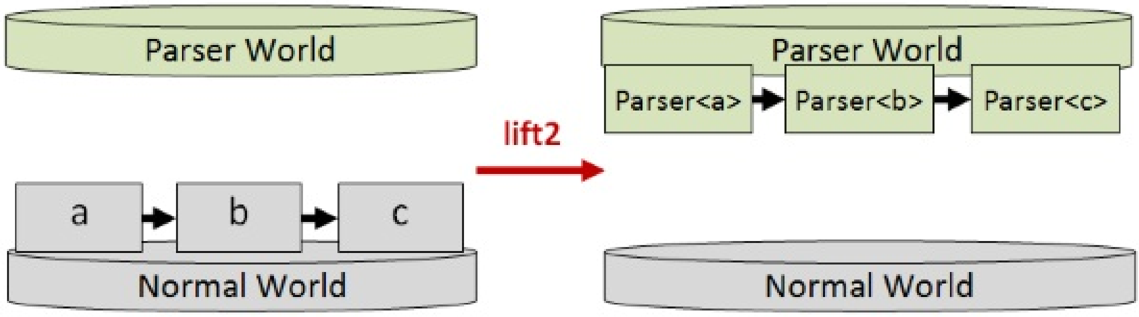
For example, we now can define a `lift2` function that will lift a two parameter function into Parser World like this:

```
// lift a two parameter function to Parser World
let lift2 f xP yP =
    returnP f <*> xP <*> yP
```

The signature of `lift2` is:

```
val lift2 :
    f:('a -> 'b -> 'c) -> Parser<'a> -> Parser<'b> -> Parser<'c>
```

Here's a diagram of `lift2`:



If you want to know more about how this works, check out my "man page" post on `lift2` or my explanation that involves the "Monadster".

Let's see some examples of using `lift2` in practice. First, lifting integer addition to addition of Parsers:

```
let addP =
    lift2 (+)
```

The signature is:

```
val addP :
    Parser<int> -> Parser<int> -> Parser<int>
```

which shows that `addP` does indeed take two `Parser<int>` parameters and returns another `Parser<int>`.

And here's the `startsWith` function being lifted to Parser World:

```
let startsWith (str:string) prefix =
    str.StartsWith(prefix)

let startsWithP =
    lift2 startsWith
```

Again, the signature of `startsWithP` is parallel to the signature of `startsWith`, but lifted to the world of Parsers.

```
val startsWith :
    str:string -> prefix:string -> bool

val startsWithP :
    Parser<string> -> Parser<string> -> Parser<bool>
```

3. `sequence` -- transforming a list of Parsers into a single Parser

We now have the tools we need to implement our sequencing combinator! The logic will be:

- Start with the list "cons" operator. This is the two-parameter function that prepends a "head" element onto a "tail" of elements to make a new list.
- Lift `cons` into the world of Parsers using `lift2`.
- We now have a a function that prepends a head `Parser` to a tail list of `Parser`s to make a new list of `Parser`s, where:
  - The head Parser is the first element in the list of parsers that has been passed in.
  - The tail is generated by calling the same function recursively with the next parser in the list.
- When the input list is empty, just return a `Parser` containing an empty list.

Here's the implementation:

```
let rec sequence parserList =
    // define the "cons" function, which is a two parameter function
    let cons head tail = head::tail

    // lift it to Parser World
    let consP = lift2 cons

    // process the list of parsers recursively
    match parserList with
    | [] ->
        returnP []
    | head::tail ->
        consP head (sequence tail)
```

The signature of `sequence` is:

```
val sequence :
    Parser<'a> list -> Parser<'a list>
```

which shows that the input is a list of `Parser`s and the output is a `Parser` containing a list of elements.

Let's test it by creating a list of three parsers, and then combining them into one:

```
let parsers = [ pchar 'A'; pchar 'B'; pchar 'C' ]
let combined = sequence parsers

run combined "ABCD"
// Success (['A'; 'B'; 'C'], "D")
```

As you can see, when we run it we get back a list of characters, one for each parser in the original list.

Implementing the `pstring` parser

At last, we can implement the parser that matches a string, which we'll call `pstring`.

The logic is:

- Convert the string into a list of characters.
- Convert each character into a `Parser<char>`.
- Use `sequence` to convert the list of `Parser<char>` into a single `Parser<char list>`.
- And finally, use `map` to convert the `Parser<char list>` into a `Parser<string>`.

Here's the code:

```
/// Helper to create a string from a list of chars
let charListToStr charList =
     String(List.toArray charList)

// match a specific string
let pstring str =
    str
    // convert to list of char
    |> List.ofSeq
    // map each char to a pchar
    |> List.map pchar
    // convert to Parser<char list>
    |> sequence
    // convert Parser<char list> to Parser<string>
    |> mapP charListToStr
```

Let's test it:

```
let parseABC = pstring "ABC"

run parseABC "ABCDE"  // Success ("ABC", "DE")
run parseABC "A|CDE"  // Failure "Expecting 'B'. Got '|'"
run parseABC "AB|DE"  // Failure "Expecting 'C'. Got '|'"
```

It works as expected. Phew!

4. `many` and `many1` -- matching a parser multiple times

Another common need is to match a particular parser as many times as you can. For example:

- When matching an integer, you want to match as many digit characters as you can.
- When matching a run of whitespace, you want to match as many whitespace characters as you can.

There are slightly different requirements for these two cases.

- When matching whitespace, it is often optional, so we want a "zero or more" matcher, which we'll call `many`.
- On the other hand, when matching digits for an integer, you want to match *at least one* digit, so we want a "one or more" matcher, which we'll call `many1`.

Before creating these, we'll define a helper function which matches a parser zero or more times. The logic is:

- Run the parser.
- If the parser returns `Failure` (and this is key) just return an empty list. That is, this function can never fail!
- If the parser succeeds:
  - Call the function recursively to get the remaining values (which could also be an empty list).
  - Then combine the first value and the remaining values.

Here's the code:

```
let rec parseZeroOrMore parser input =
    // run parser with the input
    let firstResult = run parser input
    // test the result for Failure/Success
    match firstResult with
    | Failure err ->
        // if parse fails, return empty list
        ([],input)
    | Success (firstValue,inputAfterFirstParse) ->
        // if parse succeeds, call recursively
        // to get the subsequent values
        let (subsequentValues,remainingInput) =
            parseZeroOrMore parser inputAfterFirstParse
        let values = firstValue::subsequentValues
        (values,remainingInput)
```

With this helper function, we can easily define `many` now -- it's just a wrapper over `parseZeroOrMore`:

```
/// match zero or more occurences of the specified parser
let many parser =

    let rec innerFn input =
        // parse the input -- wrap in Success as it always succeeds
        Success (parseZeroOrMore parser input)

    Parser innerFn
```

The signature of `many` shows that the output is indeed a list of values wrapped in a `Parser`:

```
val many :
    Parser<'a> -> Parser<'a list>
```

Now let's test `many`:

```
let manyA = many (pchar 'A')

// test some success cases
run manyA "ABCD"  // Success (['A'], "BCD")
run manyA "AACD"  // Success (['A'; 'A'], "CD")
run manyA "AAAD"  // Success (['A'; 'A'; 'A'], "D")

// test a case with no matches
run manyA "|BCD"  // Success ([], "|BCD")
```

Note that in the last case, even when there is nothing to match, the function succeeds.

There's nothing about `many` that restricts its use to single characters. For example, we can use it to match repetitive string sequences too:

```
let manyAB = many (pstring "AB")

run manyAB "ABCD"  // Success (["AB"], "CD")
run manyAB "ABABCD"  // Success (["AB"; "AB"], "CD")
run manyAB "ZCD"  // Success ([], "ZCD")
run manyAB "AZCD"  // Success ([], "AZCD")
```

Finally, let's implement the original example of matching whitespace:

```
let whitespaceChar = anyOf [' '; '\t'; '\n']
let whitespace = many whitespaceChar

run whitespace "ABC"  // Success ([], "ABC")
run whitespace " ABC"  // Success ([' '], "ABC")
run whitespace "\tABC"  // Success (['\t'], "ABC")
```

Defining `many1`

We can also define the "one or more" combinator `many1`, using the following logic:

- Run the parser.
- If it fails, return the failure.
- If it succeeds:
  - Call the helper function `parseZeroOrMore` to get the remaining values.
  - Then combine the first value and the remaining values.

```
/// match one or more occurences of the specified parser
let many1 parser =
    let rec innerFn input =
        // run parser with the input
        let firstResult = run parser input
        // test the result for Failure/Success
        match firstResult with
        | Failure err ->
            Failure err // failed
        | Success (firstValue,inputAfterFirstParse) ->
            // if first found, look for zeroOrMore now
            let (subsequentValues,remainingInput) =
                parseZeroOrMore parser inputAfterFirstParse
            let values = firstValue::subsequentValues
            Success (values,remainingInput)
    Parser innerFn
```

Again, the signature of `many1` shows that the output is indeed a list of values wrapped in a `Parser`:

```
val many1 :
    Parser<'a> -> Parser<'a list>
```

Now let's test `many1`:

```
// define parser for one digit
let digit = anyOf ['0'..'9']

// define parser for one or more digits
let digits = many1 digit

run digits "1ABC"  // Success (['1'], "ABC")
run digits "12BC"  // Success (['1'; '2'], "BC")
run digits "123C"  // Success (['1'; '2'; '3'], "C")
run digits "1234"  // Success (['1'; '2'; '3'; '4'], "")

run digits "ABC"   // Failure "Expecting '9'. Got 'A'"
```

As we saw in an earlier example, the last case gives a misleading error. It says "Expecting '9'" when it really should say "Expecting a digit". In the next post we'll fix this.

Parsing an integer

Using `many1`, we can create a parser for an integer. The implementation logic is:

- Create a parser for a digit.

- Use `many1` to get a list of digits.
- Using `map`, transform the result (a list of digits) into a string and then into an int.

Here's the code:

```
let pint =
    // helper
    let resultToInt digitList =
        // ignore int overflow for now
        String(List.toArray digitList) |> int

    // define parser for one digit
    let digit = anyOf ['0'..'9']

    // define parser for one or more digits
    let digits = many1 digit

    // map the digits to an int
    digits
    |> mapP resultToInt
```

And let's test it:

```
run pint "1ABC"  // Success (1, "ABC")
run pint "12BC"  // Success (12, "BC")
run pint "123C"  // Success (123, "C")
run pint "1234"  // Success (1234, "")

run pint "ABC"   // Failure "Expecting '9'. Got 'A'"
```

5. `opt` -- matching a parser zero or one time

Sometimes we only want to match a parser zero or one time. For example, the `pint` parser above does not handle negative values. To correct this, we need to be able to handle an optional minus sign.

We can define an `opt` combinator easily:

- Change the result of a specified parser to an option by mapping the result to `Some`.
- Create another parser that always returns `None`.
- Use `<|>` to choose the second ("None") parser if the first fails.

Here's the code:

```
let opt p =
    let some = p |>> Some
    let none = returnP None
    some <|> none
```

Here's an example of it in use -- we match a digit followed by an optional semicolon:

```
let digit = anyOf ['0'..'9']
let digitThenSemicolon = digit .>>. opt (pchar ';')

run digitThenSemicolon "1;"  // Success (('1', Some ';'), "")
run digitThenSemicolon "1"   // Success (('1', None), "")
```

And here is `pint` rewritten to handle an optional minus sign:

```
let pint =
    // helper
    let resultToInt (sign,charList) =
        let i = String(List.toArray charList) |> int
        match sign with
        | Some ch -> -i  // negate the int
        | None -> i

    // define parser for one digit
    let digit = anyOf ['0'..'9']

    // define parser for one or more digits
    let digits = many1 digit

    // parse and convert
    opt (pchar '-') .>>. digits
    |>> resultToInt
```

Note that the `resultToInt` helper function now needs to handle the sign option as well as the list of digits.

And here it is in action:

```
run pint "123C"   // Success (123, "C")
run pint "-123C"  // Success (-123, "C")
```

6. Throwing results away

We often want to match something in the input, but we don't care about the parsed value itself. For example:

- For a quoted string, we need to parse the quotes, but we don't need the quotes themselves.
- For a statement ending in a semicolon, we need to ensure the semicolon is there, but we don't need the semicolon itself.
- For whitespace separators, we need to ensure the whitespace is there, but we don't need the actual whitespace data.

To handle these requirements, we will define some new combinators that throw away the results of a parser:

- `p1 >>. p2` will apply `p1` and `p2` in sequence, just like `.>>.` , but throw away the result of `p1` and keep the result of `p2` .
- `p1 .>> p2` will apply `p1` and `p2` in sequence, just like `.>>.` , but keep the result of `p1` and throw away the result of `p2` .

These are easy to define -- just map over the result of `.>>.` , which is a tuple, and keep only one element of the pair.

```
/// Keep only the result of the left side parser
let (.>>) p1 p2 =
    // create a pair
    p1 .>>. p2
    // then only keep the first value
    |> mapP (fun (a,b) -> a)

/// Keep only the result of the right side parser
let (>>.) p1 p2 =
    // create a pair
    p1 .>>. p2
    // then only keep the second value
    |> mapP (fun (a,b) -> b)
```

These combinators allow us to simplify the `digitThenSemicolon` example shown earlier:

```
let digit = anyOf ['0'..'9']

// use .>> below
let digitThenSemicolon = digit .>> opt (pchar ';')

run digitThenSemicolon "1;"  // Success ('1', "")
run digitThenSemicolon "1"   // Success ('1', "")
```

You can see that the result now is the same, whether or not the semicolon was present.

How about an example with whitespace?

The following code creates a parser that looks for "AB" followed by one or more whitespace chars, followed by "CD".

```
let whitespaceChar = anyOf [' '; '\t'; '\n']
let whitespace = many1 whitespaceChar

let ab = pstring "AB"
let cd = pstring "CD"
let ab_cd = (ab .>> whitespace) .>>. cd

run ab_cd "AB \t\nCD"   // Success (("AB", "CD"), "")
```

The result contains "AB" and "CD" only. The whitespace between them has been discarded.

Introducing `between`

A particularly common requirement is to look for a parser between delimiters such as quotes or brackets.

Creating a combinator for this is trivial:

```
/// Keep only the result of the middle parser
let between p1 p2 p3 =
    p1 >>. p2 .>> p3
```

And here it is in use, to parse a quoted integer:

```
let pdoublequote = pchar '"'
let quotedInteger = between pdoublequote pint pdoublequote

run quotedInteger "\"1234\""   // Success (1234, "")
run quotedInteger "1234"       // Failure "Expecting '"'. Got '1'"
```

7. Parsing lists with separators

Another common requirement is parsing lists, seperated by something like commas or whitespace.

To implement a "one or more" list, we need to:

- First combine the separator and parser into one combined parser, but using `>>.` to throw away the separator value.

- Next, look for a list of the separator/parser combo using `many`.
- Then prefix that with the first parser and combine the results.

Here's the code:

```
/// Parses one or more occurrences of p separated by sep
let sepBy1 p sep =
    let sepThenP = sep >>. p
    p .>>. many sepThenP
    |>> fun (p,pList) -> p::pList
```

For the "zero or more" version, we can choose the empty list as an alternate if `sepBy1` does not find any matches:

```
/// Parses zero or more occurrences of p separated by sep
let sepBy p sep =
    sepBy1 p sep <|> returnP []
```

Here's some tests for `sepBy1` and `sepBy`, with results shown in the comments:

```
let comma = pchar ','
let digit = anyOf ['0'..'9']

let zeroOrMoreDigitList = sepBy digit comma
let oneOrMoreDigitList = sepBy1 digit comma

run oneOrMoreDigitList "1;"      // Success (['1'], ";")
run oneOrMoreDigitList "1,2;"    // Success (['1'; '2'], ";")
run oneOrMoreDigitList "1,2,3;"  // Success (['1'; '2'; '3'], ";")
run oneOrMoreDigitList "Z;"      // Failure "Expecting '9'. Got 'Z'"

run zeroOrMoreDigitList "1;"     // Success (['1'], ";")
run zeroOrMoreDigitList "1,2;"   // Success (['1'; '2'], ";")
run zeroOrMoreDigitList "1,2,3;" // Success (['1'; '2'; '3'], ";")
run zeroOrMoreDigitList "Z;"     // Success ([], "Z;")
```

What about `bind`?

One combinator that we *haven't* implemented so far is `bind` (or `>>=`).

If you know anything about functional programming, or have seen my talk on FP patterns, you'll know that `bind` is a powerful tool that can be used to implement many functions.

Up to this point, I thought that it would be better to show implementations for combinators such as `map` and `.>>.` that were explicit and thus, hopefully, easier to understand.

But now that we have have some experience, let's implement `bind` and see what we can do with it.

Here's the implementation of `bindP` (as I'll call it)

```
/// "bindP" takes a parser-producing function f, and a parser p
/// and passes the output of p into f, to create a new parser
let bindP f p =
    let innerFn input =
        let result1 = run p input
        match result1 with
        | Failure err ->
            // return error from parser1
            Failure err
        | Success (value1,remainingInput) ->
            // apply f to get a new parser
            let p2 = f value1
            // run parser with remaining input
            run p2 remainingInput
    Parser innerFn
```

The signature of `bindP` is:

```
val bindP :
    f:('a -> Parser<'b>) -> Parser<'a> -> Parser<'b>
```

which conforms to a standard bind signature. The input `f` is a "diagonal" function (`'a -> Parser<'b>`) and the output is a "horizontal" function (`Parser<'a> -> Parser<'b>`). See this post for more details on how `bind` works.

The infix version of `bind` is `>>=` . Note that the parameters are flipped: `f` is now the second parameter which makes it more convenient for F#'s pipeline idiom.

```
let ( >>= ) p f = bindP f p
```

Reimplementing other combinators with `bindP` and `returnP`

The combination of `bindP` and `returnP` can be used to re-implement many of the other combinators. Here are some examples:

```
let mapP f =
    bindP (f >> returnP)

let andThen p1 p2 =
    p1 >>= (fun p1Result ->
    p2 >>= (fun p2Result ->
        returnP (p1Result,p2Result) ))

let applyP fP xP =
    fP >>= (fun f ->
    xP >>= (fun x ->
        returnP (f x) ))

// (assuming "many" is defined)

let many1 p =
    p      >>= (fun head ->
    many p >>= (fun tail ->
        returnP (head::tail) ))
```

Note that the combinators that check the `Failure` path can not be implemented using `bind`. These include `orElse` and `many`.

Review

We could keep building combinators for ever, but I think we have everything we need to build a JSON parser now, so let's stop and review what we have done.

In the previous post we created these combinators:

- `.>>.` (`andThen`) applies the two parsers in sequence and returns the results in a tuple.
- `<|>` (`orElse`) applies the first parser, and if that fails, the second parsers.
- `choice` extends `orElse` to choose from a list of parsers.

And in this post we created the following additional combinators:

- `bindP` chains the result of a parser to another parser-producing function.
- `mapP` transforms the result of a parser.
- `returnP` lifts an normal value into the world of parsers.
- `applyP` allows us to lift multi-parameter functions into functions that work on Parsers.
- `lift2` uses `applyP` to lift two-parameter functions into Parser World.
- `sequence` converts a list of Parsers into a Parser containing a list.
- `many` matches zero or more occurences of the specified parser.
- `many1` matches one or more occurences of the specified parser.
- `opt` matches an optional occurrence of the specified parser.

- `.>>` keeps only the result of the left side parser.
- `>>.` keeps only the result of the right side parser.
- `between` keeps only the result of the middle parser.
- `sepBy` parses zero or more occurrences of a parser with a separator.
- `sepBy1` parses one or more occurrences of a parser with a separator.

I hope you can see why the concept of "combinators" is so powerful; given just a few basic functions, we have built up a library of useful functions quickly and concisely.

Listing of the parser library so far

Here's the complete listing for the parsing library so far -- it's about 200 lines of code now!

The source code displayed below is also available at this gist.

```
open System

/// Type that represents Success/Failure in parsing
type Result<'a> =
    | Success of 'a
    | Failure of string

/// Type that wraps a parsing function
type Parser<'T> = Parser of (string -> Result<'T * string>)

/// Parse a single character
let pchar charToMatch =
    // define a nested inner function
    let innerFn str =
        if String.IsNullOrEmpty(str) then
            Failure "No more input"
        else
            let first = str.[0]
            if first = charToMatch then
                let remaining = str.[1..]
                Success (charToMatch,remaining)
            else
                let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch first
                Failure msg
    // return the "wrapped" inner function
    Parser innerFn

/// Run a parser with some input
let run parser input =
    // unwrap parser to get inner function
    let (Parser innerFn) = parser
    // call inner function with input
    innerFn input
```

```
/// "bindP" takes a parser-producing function f, and a parser p
/// and passes the output of p into f, to create a new parser
let bindP f p =
    let innerFn input =
        let result1 = run p input
        match result1 with
        | Failure err ->
            // return error from parser1
            Failure err
        | Success (value1,remainingInput) ->
            // apply f to get a new parser
            let p2 = f value1
            // run parser with remaining input
            run p2 remainingInput
    Parser innerFn

/// Infix version of bindP
let ( >>= ) p f = bindP f p

/// Lift a value to a Parser
let returnP x =
    let innerFn input =
        // ignore the input and return x
        Success (x,input)
    // return the inner function
    Parser innerFn

/// apply a function to the value inside a parser
let mapP f =
    bindP (f >> returnP)

/// infix version of mapP
let ( <!> ) = mapP

/// "piping" version of mapP
let ( |>> ) x f = mapP f x

/// apply a wrapped function to a wrapped value
let applyP fP xP =
    fP >>= (fun f ->
    xP >>= (fun x ->
        returnP (f x) ))

/// infix version of apply
let ( <*> ) = applyP

/// lift a two parameter function to Parser World
let lift2 f xP yP =
    returnP f <*> xP <*> yP

/// Combine two parsers as "A andThen B"
let andThen p1 p2 =
```

```
    p1 >>= (fun p1Result ->
    p2 >>= (fun p2Result ->
        returnP (p1Result,p2Result) ))

/// Infix version of andThen
let ( .>>. ) = andThen

/// Combine two parsers as "A orElse B"
let orElse p1 p2 =
    let innerFn input =
        // run parser1 with the input
        let result1 = run p1 input

        // test the result for Failure/Success
        match result1 with
        | Success result ->
            // if success, return the original result
            result1

        | Failure err ->
            // if failed, run parser2 with the input
            let result2 = run p2 input

            // return parser2's result
            result2

    // return the inner function
    Parser innerFn

/// Infix version of orElse
let ( <|> ) = orElse

/// Choose any of a list of parsers
let choice listOfParsers =
    List.reduce ( <|> ) listOfParsers

/// Choose any of a list of characters
let anyOf listOfChars =
    listOfChars
    |> List.map pchar // convert into parsers
    |> choice

/// Convert a list of Parsers into a Parser of a list
let rec sequence parserList =
    // define the "cons" function, which is a two parameter function
    let cons head tail = head::tail

    // lift it to Parser World
    let consP = lift2 cons

    // process the list of parsers recursively
    match parserList with
    | [] ->
```

```
        returnP []
    | head::tail ->
        consP head (sequence tail)

/// (helper) match zero or more occurences of the specified parser
let rec parseZeroOrMore parser input =
    // run parser with the input
    let firstResult = run parser input
    // test the result for Failure/Success
    match firstResult with
    | Failure err ->
        // if parse fails, return empty list
        ([],input)
    | Success (firstValue,inputAfterFirstParse) ->
        // if parse succeeds, call recursively
        // to get the subsequent values
        let (subsequentValues,remainingInput) =
            parseZeroOrMore parser inputAfterFirstParse
        let values = firstValue::subsequentValues
        (values,remainingInput)

/// matches zero or more occurences of the specified parser
let many parser =
    let rec innerFn input =
        // parse the input -- wrap in Success as it always succeeds
        Success (parseZeroOrMore parser input)

    Parser innerFn

/// matches one or more occurences of the specified parser
let many1 p =
    p      >>= (fun head ->
    many p >>= (fun tail ->
        returnP (head::tail) ))

/// Parses an optional occurrence of p and returns an option value.
let opt p =
    let some = p |>> Some
    let none = returnP None
    some <|> none

/// Keep only the result of the left side parser
let (.>>) p1 p2 =
    // create a pair
    p1 .>>. p2
    // then only keep the first value
    |> mapP (fun (a,b) -> a)

/// Keep only the result of the right side parser
let (>>.) p1 p2 =
    // create a pair
    p1 .>>. p2
    // then only keep the second value
```

```
    |> mapP (fun (a,b) -> b)

/// Keep only the result of the middle parser
let between p1 p2 p3 =
    p1 >>. p2 .>> p3

/// Parses one or more occurrences of p separated by sep
let sepBy1 p sep =
    let sepThenP = sep >>. p
    p .>>. many sepThenP
    |>> fun (p,pList) -> p::pList

/// Parses zero or more occurrences of p separated by sep
let sepBy p sep =
    sepBy1 p sep <|> returnP []
```

Summary

In this post, we have built on the basic parsing code from last time to create a library of a 15 or so combinators that can be combined to parse almost anything.

Soon, we'll use them to build a JSON parser, but before that, let's pause and take time to clean up the error messages. That will be the topic of the next post.

The source code for this post is available at this gist.

Improving the parser library

UPDATE: Slides and video from my talk on this topic

In this series, we are looking at how applicative parsers and parser combinators work.

- In the first post, we created the foundations of a parsing library.
- In the second post, we built out the library with many other useful combinators.
- In this post, we'll rework the library to provide more helpful error messages.

---

1. Labelling a Parser

In some of the failing code examples from earlier posts, we got confusing errors:

```
let parseDigit = anyOf ['0'..'9']
run parseDigit "|ABC"  // Failure "Expecting '9'. Got '|'"
```

`parseDigit` is defined as a choice of digit characters, so when the last choice (`'9'`) fails, that is the error message we receive.

But that message is quite confusing. What we *really* want is to receive is an error that mentions "digit", something like: `Failure "Expecting digit. Got '|'"` .

That is, what we need is a way of labeling parsers with a word like "digit" and then showing that label when a failure occurs.

As a reminder, this is how the `Parser` type was defined in earlier posts:

```
type Parser<'a> = Parser of (string -> Result<'a * string>)
```

In order to add a label, we need to change it into a record structure:

```
type ParserLabel = string

/// A Parser structure has a parsing function & label
type Parser<'a> = {
    parseFn : (string -> Result<'a * string>)
    label:  ParserLabel
    }
```

The record contains two fields: the parsing function (`parseFn`) and the `label` .

One problem is that the label is in the parser itself, but not in the `Result` , which means that clients will not know how to display the label along with the error.

So let's add it to the `Failure` case of `Result` as well, in addition to the error message:

```
// Aliases
type ParserLabel = string
type ParserError = string

type Result<'a> =
    | Success of 'a
    | Failure of ParserLabel * ParserError
```

And while we are at it, let's define a helper function to display the result of a parse:

```
let printResult result =
    match result with
    | Success (value,input) ->
        printfn "%A" value
    | Failure (label,error) ->
        printfn "Error parsing %s\n%s" label error
```

Updating the code

With this change to the definition of `Parser` and `Result` , we have to change some of the basic functions, such as `bindP` :

```
/// "bindP" takes a parser-producing function f, and a parser p
/// and passes the output of p into f, to create a new parser
let bindP f p =
    let label = "unknown"           // <====== "label" is new!
    let innerFn input =
        ...
        match result1 with
        | Failure (label,err) ->    // <====== "label" is new!
            ...
        | Success (value1,remainingInput) ->
            ...
    {parseFn=innerFn; label=label}  // <====== "parseFn" and "label" are new!
```

We have to make similar changes to `returnP` , `orElse` , and `many` . For the complete code, see the gist linked to below.

Updating the label

When we use a combinator to build a new compound parser, we will often want to assign a new label to it. In order to do this, we replace the original `parseFn` with another one that returns the new label.

Here's the code:

```
/// Update the label in the parser
let setLabel parser newLabel =
    // change the inner function to use the new label
    let newInnerFn input =
        let result = parser.parseFn input
        match result with
        | Success s ->
            // if Success, do nothing
            Success s
        | Failure (oldLabel,err) ->
            // if Failure, return new label
            Failure (newLabel,err)        // <====== use newLabel here
    // return the Parser
    {parseFn=newInnerFn; label=newLabel}  // <====== use newLabel here
```

And let's create an infix version of this called `<?>`:

```
/// infix version of setLabel
let ( <?> ) = setLabel
```

Let's test our new toy!

```
let parseDigit_WithLabel =
    anyOf ['0'..'9']
    <?> "digit"

run parseDigit_WithLabel "|ABC"
|> printResult
```

And the output is:

```
Error parsing digit
Unexpected '|'
```

The error message is now `Error parsing digit` rather than `Expecting '9'`. Much better!

Setting default labels:

We can also set the default labels for certain combinators such as `andThen` and `orElse` based on the inputs:

```
/// Combine two parsers as "A andThen B"
let andThen p1 p2 =
    let label = sprintf "%s andThen %s" (getLabel p1) (getLabel p2)
    p1 >>= (fun p1Result ->
    p2 >>= (fun p2Result ->
        returnP (p1Result,p2Result) ))
    <?> label         // <====== provide a custom label

// combine two parsers as "A orElse B"
let orElse parser1 parser2 =
    // construct a new label
    let label =       // <====== provide a custom label
        sprintf "%s orElse %s" (getLabel parser1) (getLabel parser2)


    let innerFn input =
       ... etc ...

/// choose any of a list of characters
let anyOf listOfChars =
    let label = sprintf "any of %A" listOfChars
    listOfChars
    |> List.map pchar
    |> choice
    <?> label         // <====== provide a custom label
```

---

2. Replacing "pchar" with "satisfy"

One thing that has bothered me about all the implementations so far is `pchar`, the basic primitive that all the other functions have built on.

I don't like that it is so tightly coupled to the input model. What happens if we want to parse bytes from a binary format, or other kinds of input. All the combinators other than `pchar` are loosely coupled. If we could decouple `pchar` as well, we would be set up for parsing *any* stream of tokens, and that would make me happy!

At this point, I'll repeat one of my favorite FP slogans: "parameterize all the things!" In the case of `pchar`, we'll remove the `charToMatch` parameter and replace it with a function -- a predicate. We'll call the new function `satisfy`:

```
/// Match an input token if the predicate is satisfied
let satisfy predicate label =
    let innerFn input =
        if String.IsNullOrEmpty(input) then
            Failure (label,"No more input")
        else
            let first = input.[0]
            if predicate first then      // <====== use predicate here
                let remainingInput = input.[1..]
                Success (first,remainingInput)
            else
                let err = sprintf "Unexpected '%c'" first
                Failure (label,err)
    // return the parser
    {parseFn=innerFn;label=label}
```

Other than the parameters, the only thing that has changed from the `pchar` implementation is this one line:

```
let satisfy predicate label =
    ...
    if predicate first then
    ...
```

With `satisfy` available, we can rewrite `pchar` :

```
/// parse a char
let pchar charToMatch =
    let predicate ch = (ch = charToMatch)
    let label = sprintf "%c" charToMatch
    satisfy predicate label
```

Note that we are setting the label to be the `charToMatch` . This refactoring would not have been as convenient before, because we didn't have the concept of "labels" yet, and so `pchar` would not have been able to return a useful error message.

The `satisfy` function also lets us write more efficient versions of other parsers. For example, parsing a digit looked like this originally:

```
/// parse a digit
let digitChar =
    anyOf ['0'..'9']
```

But now we can rewrite it using a predicate directly, making it a lot more efficient:

```
/// parse a digit
let digitChar =
    let predicate = Char.IsDigit
    let label = "digit"
    satisfy predicate label
```

Similarly, we can create a more efficient whitespace parser too:

```
/// parse a whitespace char
let whitespaceChar =
    let predicate = Char.IsWhiteSpace
    let label = "whitespace"
    satisfy predicate label
```

3. Adding position and context to error messages

Another way to improve the error messages is to show the line and column that the error occurred on.

Obviously, for simple one-liners, keeping track of the error location is not a problem, but when you are parsing a 100 line JSON file, it will be very helpful.

In order to track the line and column we are going to have to abandon the simple `string` input and replace it with something more complex, so let's start with that.

Defining a input that tracks position

First, we will need a `Position` type to store the line and column, with helper functions to increment one column and one line:

```
type Position = {
    line : int
    column : int
}

/// define an initial position
let initialPos = {line=0; column=0}

/// increment the column number
let incrCol pos =
    {pos with column=pos.column + 1}

/// increment the line number and set the column to 0
let incrLine pos =
    {line=pos.line + 1; column=0}
```

Next, we'll need to combine the input string with a position into a single "input state" type. Since we are line oriented, we can make our lives easier and store the input string as a array of lines rather than as one giant string:

```
/// Define the current input state
type InputState = {
    lines : string[]
    position : Position
}
```

We will also need a way to convert a string into a initial `InputState`:

```
/// Create a new InputState from a string
let fromStr str =
    if String.IsNullOrEmpty(str) then
        {lines=[||]; position=initialPos}
    else
        let separators = [| "\r\n"; "\n" |]
        let lines = str.Split(separators, StringSplitOptions.None)
        {lines=lines; position=initialPos}
```

Finally, and most importantly, we need a way to read the next character from the input -- let's call it `nextChar`.

We know what the input for `nextChar` will be (an `InputState`) but what should the output look like?

- If the input is at the end, we need a way to indicate that there is no next character, so in that case return `None`.
- Therefore in the case when a character is available, we will return `Some`.

- In addition, the input state will have changed because the column (or line) will have been incremented as well.

So, putting this together, the input for `nextChar` is an `InputState` and the output is a pair `char option * InputState`.

The logic for returning the next char will be as follows then:

- If we are at the last character of the input, return EOF (`None`) and don't change the state.
- If the current column is *not* at the end of a line, return the character at that position and change the state by incrementing the column position.
- If the current column *is* at the end of a line, return a newline character and change the state by incrementing the line position.

Here's the code:

```
// return the current line
let currentLine inputState =
    let linePos = inputState.position.line
    if linePos < inputState.lines.Length then
        inputState.lines.[linePos]
    else
        "end of file"

/// Get the next character from the input, if any
/// else return None. Also return the updated InputState
/// Signature: InputState -> InputState * char option
let nextChar input =
    let linePos = input.position.line
    let colPos = input.position.column
    // three cases
    // 1) if line >= maxLine ->
    //       return EOF
    // 2) if col less than line length ->
    //       return char at colPos, increment colPos
    // 3) if col at line length ->
    //       return NewLine, increment linePos

    if linePos >= input.lines.Length then
        input, None
    else
        let currentLine = currentLine input
        if colPos < currentLine.Length then
            let char = currentLine.[colPos]
            let newPos = incrCol input.position
            let newState = {input with position=newPos}
            newState, Some char
        else
            // end of line, so return LF and move to next line
            let char = '\n'
            let newPos = incrLine input.position
            let newState = {input with position=newPos}
            newState, Some char
```

Unlike the earlier `string` implementation, the underlying array of lines is never altered or copied -- only the position is changed. This means that making a new state each time the position changes should be reasonably efficient, because the text is shared everywhere.

Let's quickly test that the implementation works. We'll create a helper function `readAllChars` and then see what it returns for different inputs:

```
let rec readAllChars input =
    [
        let remainingInput,charOpt = nextChar input
        match charOpt with
        | None ->
            // end of input
            ()
        | Some ch ->
            // return first character
            yield ch
            // return the remaining characters
            yield! readAllChars remainingInput
    ]
```

Here it is with some example inputs:

```
fromStr "" |> readAllChars       // []
fromStr "a" |> readAllChars      // ['a'; '\n']
fromStr "ab" |> readAllChars     // ['a'; 'b'; '\n']
fromStr "a\nb" |> readAllChars   // ['a'; '\n'; 'b'; '\n']
```

Note that the implementation returns a newline at the end of the input, even if the input doesn't have one. I think that this is a feature, not a bug!

Changing the parser to use the input

We now need to change the `Parser` type again.

To start with, the `Failure` case needs to return some kind of data that indicates the position, so we can show it in an error message.

We could just use the `InputState` as is, but let's be good and define a new type specially for this use, called `ParserPosition` :

```
/// Stores information about the parser position for error messages
type ParserPosition = {
    currentLine : string
    line : int
    column : int
    }
```

We'll need some way to convert a `InputState` into a `ParserPosition` :

```
let parserPositionFromInputState (inputState:Input) = {
    currentLine = TextInput.currentLine inputState
    line = inputState.position.line
    column = inputState.position.column
    }
```

And finally, we can update the `Result` type to include `ParserPosition` :

```
// Result type
type Result<'a> =
    | Success of 'a
    | Failure of ParserLabel * ParserError * ParserPosition
```

In addition, the `Parser` type needs to change from `string` to `InputState` :

```
type Input = TextInput.InputState  // type alias

/// A Parser structure has a parsing function & label
type Parser<'a> = {
    parseFn : (Input -> Result<'a * Input>)
    label:  ParserLabel
    }
```

With all this extra information available, the `printResult` function can be enhanced to print the text of the current line, along with a caret where the error is:

```
let printResult result =
    match result with
    | Success (value,input) ->
        printfn "%A" value
    | Failure (label,error,parserPos) ->
        let errorLine = parserPos.currentLine
        let colPos = parserPos.column
        let linePos = parserPos.line
        let failureCaret = sprintf "%*s^%s" colPos "" error
        printfn "Line:%i Col:%i Error parsing %s\n%s\n%s" linePos colPos label errorLi
ne failureCaret
```

Let's test `printResult` with a dummy error value:

```
let exampleError =
    Failure ("identifier", "unexpected |",
             {currentLine = "123 ab|cd"; line=1; column=6})

printResult exampleError
```

The output is shown below:

```
Line:1 Col:6 Error parsing identifier
123 ab|cd
      ^unexpected |
```

Much nicer than before!

Fixing up the `run` function

The `run` function now needs to take an `InputState` not a string. But we also want the convenience of running against string input, so let's create two `run` functions, one that takes an `InputState` and one that takes a `string` :

```
/// Run the parser on a InputState
let runOnInput parser input =
    // call inner function with input
    parser.parseFn input

/// Run the parser on a string
let run parser inputStr =
    // call inner function with input
    runOnInput parser (TextInput.fromStr inputStr)
```

Fixing up the combinators

We now have three items in the `Failure` case rather than two. This breaks some code but is easy to fix. I'm tempted to create a special `ParserError` type so that it never happens again, but for now, I'll just fix up the errors.

Here's a new version of `satisfy` :

```
/// Match an input token if the predicate is satisfied
let satisfy predicate label =
    let innerFn input =
        let remainingInput,charOpt = TextInput.nextChar input
        match charOpt with
        | None ->
            let err = "No more input"
            let pos = parserPositionFromInputState input
            //Failure (label,err)     // <====== old version
            Failure (label,err,pos)   // <====== new version
        | Some first ->
            if predicate first then
                Success (first,remainingInput)
            else
                let err = sprintf "Unexpected '%c'" first
                let pos = parserPositionFromInputState input
                //Failure (label,err)     // <====== old version
                Failure (label,err,pos)   // <====== new version
    // return the parser
    {parseFn=innerFn;label=label}
```

Note that the failure case code is now `Failure (label,err,pos)` where the parser position is built from the input state.

And here is `bindP` :

```
/// "bindP" takes a parser-producing function f, and a parser p
/// and passes the output of p into f, to create a new parser
let bindP f p =
    let label = "unknown"
    let innerFn input =
        let result1 = runOnInput p input
        match result1 with
        | Failure (label,err,pos) ->     // <====== new with pos
            // return error from parser1
            Failure (label,err,pos)
        | Success (value1,remainingInput) ->
            // apply f to get a new parser
            let p2 = f value1
            // run parser with remaining input
            runOnInput p2 remainingInput
    {parseFn=innerFn; label=label}
```

We can fix up the other functions in the same way.

Testing the positional errors

Let's test with a real parser now:

```
let parseAB =
    pchar 'A' .>>. pchar 'B'
    <?> "AB"

run parseAB "A|C"
|> printResult
```

And the output is:

```
// Line:0 Col:1 Error parsing AB
// A|C
//   ^Unexpected '|'
```

Excellent! I think we can stop now.

4. Adding some standard parsers to the library

In the previous posts, we've built parsers for strings and ints in passing, but now let's add them to the core library, so that clients don't have to reinvent the wheel.

These parsers are based on those in the the FParsec library.

Let's start with some string-related parsers. I will present them without comment -- I hope that the code is self-explanatory by now.

```
/// parse a char
let pchar charToMatch =
    // label is just the character
    let label = sprintf "%c" charToMatch

    let predicate ch = (ch = charToMatch)
    satisfy predicate label

/// Choose any of a list of characters
let anyOf listOfChars =
    let label = sprintf "anyOf %A" listOfChars
    listOfChars
    |> List.map pchar // convert into parsers
    |> choice
    <?> label

/// Convert a list of chars to a string
let charListToStr charList =
    String(List.toArray charList)

/// Parses a sequence of zero or more chars with the char parser cp.
/// It returns the parsed chars as a string.
let manyChars cp =
    many cp
    |>> charListToStr

/// Parses a sequence of one or more chars with the char parser cp.
/// It returns the parsed chars as a string.
let manyChars1 cp =
    many1 cp
    |>> charListToStr

/// parse a specific string
let pstring str =
    // label is just the string
    let label = str

    str
    // convert to list of char
    |> List.ofSeq
    // map each char to a pchar
    |> List.map pchar
    // convert to Parser<char list>
    |> sequence
    // convert Parser<char list> to Parser<string>
    |> mapP charListToStr
    <?> label
```

Let's test `pstring`, for example:

```
run (pstring "AB") "ABC"
|> printResult
// Success
// "AB"

run (pstring "AB") "A|C"
|> printResult
// Line:0 Col:1 Error parsing AB
// A|C
//  ^Unexpected '|'
```

Whitespace parsers

Whitespace is important in parsing, even if we do end up mostly throwing it away!

```
/// parse a whitespace char
let whitespaceChar =
    let predicate = Char.IsWhiteSpace
    let label = "whitespace"
    satisfy predicate label

/// parse zero or more whitespace char
let spaces = many whitespaceChar

/// parse one or more whitespace char
let spaces1 = many1 whitespaceChar
```

And here's some whitespace tests:

```
run spaces " ABC"
|> printResult
// [' ']

run spaces "A"
|> printResult
// []

run spaces1 " ABC"
|> printResult
// [' ']

run spaces1 "A"
|> printResult
// Line:0 Col:0 Error parsing many1 whitespace
// A
// ^Unexpected 'A'
```

Numeric parsers

Finally, we need a parser for ints and floats.

```
/// parse a digit
let digitChar =
    let predicate = Char.IsDigit
    let label = "digit"
    satisfy predicate label

// parse an integer
let pint =
    let label = "integer"

    // helper
    let resultToInt (sign,digits) =
        let i = digits |> int  // ignore int overflow for now
        match sign with
        | Some ch -> -i  // negate the int
        | None -> i

    // define parser for one or more digits
    let digits = manyChars1 digitChar

    // an "int" is optional sign + one or more digits
    opt (pchar '-') .>>. digits
    |> mapP resultToInt
    <?> label

// parse a float
let pfloat =
    let label = "float"

    // helper
    let resultToFloat (((sign,digits1),point),digits2) =
        let fl = sprintf "%s.%s" digits1 digits2 |> float
        match sign with
        | Some ch -> -fl  // negate the float
        | None -> fl

    // define parser for one or more digits
    let digits = manyChars1 digitChar

    // a float is sign, digits, point, digits (ignore exponents for now)
    opt (pchar '-') .>>. digits .>>. pchar '.' .>>. digits
    |> mapP resultToFloat
    <?> label
```

And some tests:

```
run pint "-123Z"
|> printResult
// -123

run pint "-Z123"
|> printResult
// Line:0 Col:1 Error parsing integer
// -Z123
//  ^Unexpected 'Z'

run pfloat "-123.45Z"
|> printResult
// -123.45

run pfloat "-123Z45"
|> printResult
// Line:0 Col:4 Error parsing float
// -123Z45
//     ^Unexpected 'Z'
```

5. Backtracking

One more topic that we should discuss is "backtracking".

Let's say that you have two parsers: one to match the string `A-1` and and another to match the string `A-2` . If the input is `A-2` then the first parser will fail at the third character and the second parser will be attempted.

Now the second parser must start at the *beginning* of the original sequence of characters, not at the third character. That is, we need to undo the current position in the input stream and go back to the first position.

If we were using a mutable input stream then this might be a tricky problem, but thankfully we are using immutable data, and so "undoing" the position just means using the original input value. And of course, this is exactly what combinators such as `orElse` (`<|>`) do.

In other words, we get backtracking "for free" when we use immutable input state. Yay!

Sometimes however, we *don't* want to backtrack. For example, let's say we have these parsers:

- let `forExpression` = the "for" keyword, then an identifier, then the "in" keyword, etc.
- let `ifExpression` = the "if" keyword, then an identifier, then the "then" keyword, etc.

and we then create a combined expression parser that chooses between them:

- let `expression` = `forExpression <|> ifExpression`

Now, if the input stream is `for &&& in something` then the `forExpression` parser will error when it hits the sequence `&&&`, because it is expecting a valid identifier. At this point we *don't* want to backtrack and try the `ifExpression` -- we want to show an error such as "identifier expected after 'for'".

The rule then is that: *if* input has been consumed successfully (in this case, the `for` keyword was matched successfully) then do *not* backtrack.

We're not going to implement this rule in our simple library, but a proper library like FParsec does implement this and also has support for bypassing it when needed.

Listing of the final parser library

The parsing library is up to 500 lines of code now, so I won't show it here. You can see it at this gist.

Summary

In this post, we added better error handling and some more parsers.

Now we have everything we need to build a JSON parser! That will be the topic of the next post.

The source code for this post is available at this gist.

Writing a JSON parser from scratch

UPDATE: Slides and video from my talk on this topic

In this series, we are looking at how applicative parsers and parser combinators work.

- In the first post, we created the foundations of a parsing library.
- In the second post, we built out the library with many other useful combinators.
- In the third post, we improved the error messages.
- In this last post, we'll use the library we've written to build a JSON parser.

First, before we do anything else, we need to load the parser library script that we developed over the last few posts, and then open the `ParserLibrary` namespace:

```
#load "ParserLibrary.fsx"

open System
open ParserLibrary
```

You can download `ParserLibrary.fsx` from here.

1. Building a model to represent the JSON spec

The JSON spec is available at json.org. I'll paraphase it here:

- A `value` can be a `string` or a `number` or a `bool` or `null` or an `object` or an `array` .
  - These structures can be nested.
- A `string` is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes.
- A `number` is very much like a C or Java number, except that the octal and hexadecimal formats are not used.
- A `boolean` is the literal `true` or `false`
- A `null` is the literal `null`
- An `object` is an unordered set of name/value pairs.
  - An object begins with `{` (left brace) and ends with `}` (right brace).
  - Each name is followed by `:` (colon) and the name/value pairs are separated by `,` (comma).

- An `array` is an ordered collection of values.
    - An array begins with `[` (left bracket) and ends with `]` (right bracket).
    - Values are separated by `,` (comma).
- Whitespace can be inserted between any pair of tokens.

In F#, this definition can be modelled naturally as:

```
type JValue =
    | JString of string
    | JNumber of float
    | JBool   of bool
    | JNull
    | JObject of Map<string, Json>
    | JArray  of Json list
```

So the goal of our JSON parser is:

- Given a string, we want to output a `JValue` value.

2. Getting started with `Null` and `Bool`

Let's start with the simplest tasks -- parsing the literal values for null and the booleans.

Parsing Null

Parsing the `null` literal is trivial. The logic will be:

- Match the string "null".
- Map the result to the `JNull` case.

Here's the code:

```
let jNull =
    pstring "null"
    |>> (fun _ -> JNull)  // map to JNull
    <?> "null"            // give it a label
```

Note that we don't actually care about the value returned by the parser because we know in advance that it is going to be "null"!

This is a common situation, so let's write a little utility function, `>>%` to make this look nicer:

```
// applies the parser p, ignores the result, and returns x.
let (>>%) p x =
    p |>> (fun _ -> x)
```

Now we can rewrite `jNull` as follows:

```
let jNull =
    pstring "null"
    >>% JNull   // using new utility combinator
    <?> "null"
```

Let's test:

```
run jNull "null"
// Success: JNull

run jNull "nulp" |> printResult
// Line:0 Col:3 Error parsing null
// nulp
//    ^Unexpected 'p'
```

That looks good. Let's try another one!

Parsing Bool

The bool parser will be similar to null:

- Create a parser to match "true".
- Create a parser to match "false".
- And then choose between them using `<|>`.

Here's the code:

```
let jBool =
    let jtrue =
        pstring "true"
        >>% JBool true   // map to JBool
    let jfalse =
        pstring "false"
        >>% JBool false  // map to JBool

    // choose between true and false
    jtrue <|> jfalse
    <?> "bool"           // give it a label
```

And here are some tests:

```
run jBool "true"
// Success: JBool true

run jBool "false"
// Success: JBool false

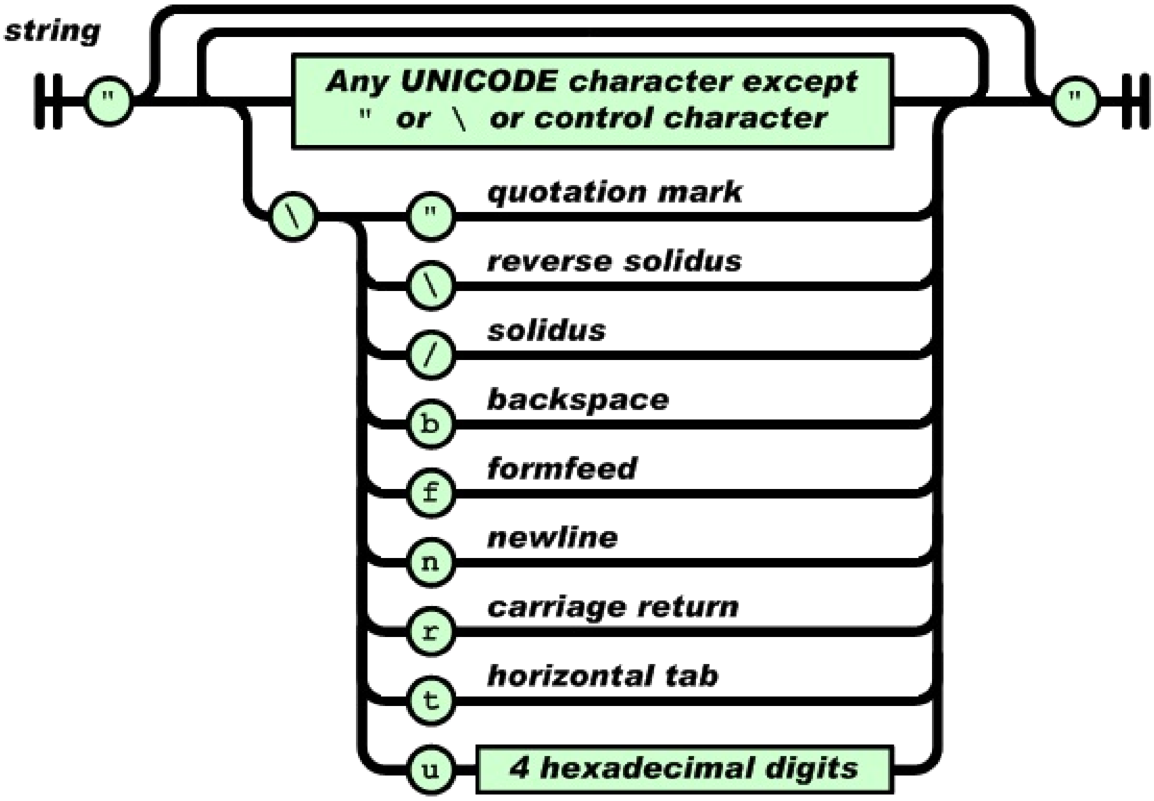
run jBool "truX" |> printResult
// Line:0 Col:0 Error parsing bool
// truX
// ^Unexpected 't'
```

Note that the error is misleading due to the backtracking issue discussed in the previous post. Since "true" failed, it is trying to parse "false" now, and "t" is an unexpected character.

3. Parsing `String`

Now for something more complicated -- strings.

The spec for string parsing is available as a "railway diagram" like this:



All diagrams sourced from json.org.

To build a parser from a diagram like this, we work from the bottom up, building small "primitive" parsers which we then combine into larger ones.

Let's start with "any unicode character other than quote and backslash". We have a simple condition to test, so we can just use the `satisfy` function:

```
let jUnescapedChar =
    let label = "char"
    satisfy (fun ch -> ch <> '\\' && ch <> '\"') label
```

We can test it immediately:

```
run jUnescapedChar "a"   // Success 'a'

run jUnescapedChar "\\" |> printResult
// Line:0 Col:0 Error parsing char
// \
// ^Unexpected '\'
```

Ok, good.

Escaped characters

Now what about the next case, the escaped characters?

In this case we have a list of strings to match (`"\""` , `"\n"` , etc) and for each of these, a character to use as the result.

The logic will be:

- First define a list of pairs in the form `(stringToMatch, resultChar)` .
- For each of these, build a parser using `pstring stringToMatch >>% resultChar)` .
- Finally, combine all these parsers together using the `choice` function.

Here's the code:

```
/// Parse an escaped char
let jEscapedChar =
    [
    // (stringToMatch, resultChar)
    ("\\\"",'\"')      // quote
    ("\\\\",'\\')      // reverse solidus
    ("\\/",'/')        // solidus
    ("\\b",'\b')       // backspace
    ("\\f",'\f')       // formfeed
    ("\\n",'\n')       // newline
    ("\\r",'\r')       // cr
    ("\\t",'\t')       // tab
    ]
    // convert each pair into a parser
    |> List.map (fun (toMatch,result) ->
        pstring toMatch >>% result)
    // and combine them into one
    |> choice
    <?> "escaped char" // set label
```

And again, let's test it immediately:

```
run jEscapedChar "\\\\" // Success '\'
run jEscapedChar "\\t"  // Success '\009'

run jEscapedChar "a" |> printResult
// Line:0 Col:0 Error parsing escaped char
// a
// ^Unexpected 'a'
```

It works nicely!

Unicode characters

The final case is the parsing of unicode characters with hex digits.

The logic will be:

- First define the primitives for `backslash` , `u` and `hexdigit` .
- Combine them together, using four `hexdigit` s.
- The output of the parser will be a nested, ugly tuple, so we need a helper function to convert the digits to an int, and then a char.

Here's the code:

```
/// Parse a unicode char
let jUnicodeChar =

    // set up the "primitive" parsers
    let backslash = pchar '\\'
    let uChar = pchar 'u'
    let hexdigit = anyOf (['0'..'9'] @ ['A'..'F'] @ ['a'..'f'])

    // convert the parser output (nested tuples)
    // to a char
    let convertToChar (((h1,h2),h3),h4) =
        let str = sprintf "%c%c%c%c" h1 h2 h3 h4
        Int32.Parse(str,Globalization.NumberStyles.HexNumber) |> char

    // set up the main parser
    backslash  >>. uChar >>. hexdigit .>>. hexdigit .>>. hexdigit .>>. hexdigit
    |>> convertToChar
```

And let's test with a smiley face -- `\u263A` .

```
run jUnicodeChar "\\u263A"
```

The complete `String` parser

Putting it all together now:

- Define a primitive for `quote`
- Define a `jchar` as a choice between `jUnescapedChar` , `jEscapedChar` , and `jUnicodeChar` .
- The whole parser is then zero or many `jchar` between two quotes.

```
let quotedString =
    let quote = pchar '\"' <?> "quote"
    let jchar = jUnescapedChar <|> jEscapedChar <|> jUnicodeChar

    // set up the main parser
    quote >>. manyChars jchar .>> quote
```

One more thing, which is to wrap the quoted string in a `JString` case and give it a label:

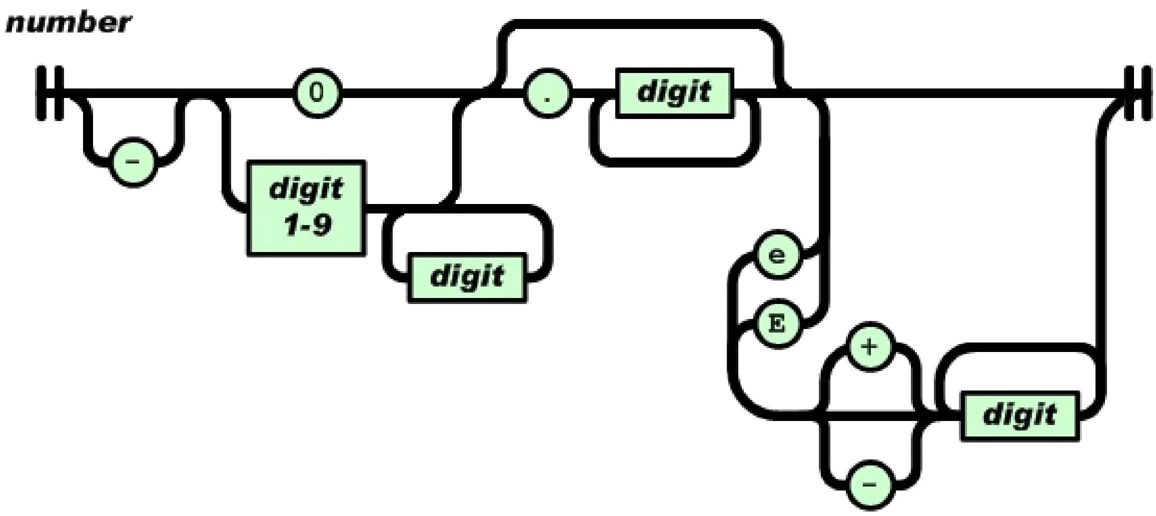
```
/// Parse a JString
let jString =
    // wrap the string in a JString
    quotedString
    |>> JString           // convert to JString
    <?> "quoted string"   // add label
```

Let's test the complete `jString` function:

```
run jString "\"\""    // Success ""
run jString "\"a\""   // Success "a"
run jString "\"ab\""  // Success "ab"
run jString "\"ab\\tde\""      // Success "ab\tde"
run jString "\"ab\\u263Ade\""  // Success "ab?de"
```

4. Parsing `Number`

The "railway diagram" for Number parsing is:



Again, we'll work bottom up. Let's start with the most primitive components, the single chars and digits:

```
let optSign = opt (pchar '-')

let zero = pstring "0"

let digitOneNine =
    satisfy (fun ch -> Char.IsDigit ch && ch <> '0') "1-9"

let digit =
    satisfy (fun ch -> Char.IsDigit ch ) "digit"

let point = pchar '.'

let e = pchar 'e' <|> pchar 'E'

let optPlusMinus = opt (pchar '-' <|> pchar '+')
```

Now let's build the "integer" part of the number. This is either:

- The digit zero, or,
- A `nonZeroInt`, which is a `digitOneNine` followed by zero or more normal digits.

```
let nonZeroInt =
    digitOneNine .>>. manyChars digit
    |>> fun (first,rest) -> string first + rest

let intPart = zero <|> nonZeroInt
```

Note that, for the `nonZeroInt` parser, we have to combine the output of `digitOneNine` (a char) with `manyChars digit` (a string) so a simple map function is needed.

The optional fractional part is a decimal point followed by one or more digits:

```
let fractionPart = point >>. manyChars1 digit
```

And the exponent part is an `e` followed by an optional sign, followed by one or more digits:

```
let exponentPart = e >>. optPlusMinus .>>. manyChars1 digit
```

With these components, we can assemble the whole number:

```
optSign .>>. intPart .>>. opt fractionPart .>>. opt exponentPart
|>> convertToJNumber
<?> "number"   // add label
```

We haven't defined `convertToJNumber` yet though. This function will take the four-tuple output by the parser and convert it into a float.

Now rather than writing custom float logic, we're going to be lazy and let the .NET framework to the conversion for us! That is, each of the components will be turned into a string, concatenated, and the whole string parsed into a float.

The problem is that some of the components (like the sign and exponent) are optional. Let's write a helper that converts an option to a string using a passed in function, but if the option is `None` return the empty string.

I'm going to call it `|>?` but it doesn't really matter because it is only used locally within the `jNumber` parser.

```
// utility function to convert an optional value to a string, or "" if missing
let ( |>? ) opt f =
    match opt with
    | None -> ""
    | Some x -> f x
```

Now we can create `convertToJNumber` :

- The sign is converted to a string.
- The fractional part is converted to a string, prefixed with a decimal point.
- The exponent part is converted to a string, with the sign of the exponent also being converted to a string.

```
let convertToJNumber (((optSign,intPart),fractionPart),expPart) =
    // convert to strings and let .NET parse them! - crude but ok for now.

    let signStr =
        optSign
        |>? string   // e.g. "-"

    let fractionPartStr =
        fractionPart
        |>? (fun digits -> "." + digits )  // e.g. ".456"

    let expPartStr =
        expPart
        |>? fun (optSign, digits) ->
            let sign = optSign |>? string
            "e" + sign + digits          // e.g. "e-12"

    // add the parts together and convert to a float, then wrap in a JNumber
    (signStr + intPart + fractionPartStr + expPartStr)
    |> float
    |> JNumber
```

It's pretty crude, and converting things to strings can be slow, so feel free to write a better version.

With that, we have everything we need for the complete `jNumber` function:

```
/// Parse a JNumber
let jNumber =

    // set up the "primitive" parsers
    let optSign = opt (pchar '-')

    let zero = pstring "0"

    let digitOneNine =
        satisfy (fun ch -> Char.IsDigit ch && ch <> '0') "1-9"

    let digit =
        satisfy (fun ch -> Char.IsDigit ch ) "digit"

    let point = pchar '.'

    let e = pchar 'e' <|> pchar 'E'

    let optPlusMinus = opt (pchar '-' <|> pchar '+')

    let nonZeroInt =
        digitOneNine .>>. manyChars digit
        |>> fun (first,rest) -> string first + rest
```

```
    let intPart = zero <|> nonZeroInt

    let fractionPart = point >>. manyChars1 digit

    let exponentPart = e >>. optPlusMinus .>>. manyChars1 digit

    // utility function to convert an optional value to a string, or "" if missing
    let ( |>? ) opt f =
        match opt with
        | None -> ""
        | Some x -> f x

    let convertToJNumber (((optSign,intPart),fractionPart),expPart) =
        // convert to strings and let .NET parse them! - crude but ok for now.

        let signStr =
            optSign
            |>? string   // e.g. "-"

        let fractionPartStr =
            fractionPart
            |>? (fun digits -> "." + digits )  // e.g. ".456"

        let expPartStr =
            expPart
            |>? fun (optSign, digits) ->
                let sign = optSign |>? string
                "e" + sign + digits          // e.g. "e-12"

        // add the parts together and convert to a float, then wrap in a JNumber
        (signStr + intPart + fractionPartStr + expPartStr)
        |> float
        |> JNumber

    // set up the main parser
    optSign .>>. intPart .>>. opt fractionPart .>>. opt exponentPart
    |>> convertToJNumber
    <?> "number"   // add label
```

It's a bit long-winded, but each component follows the spec, so I think it is still quite readable.

Let's start testing it:

```
run jNumber "123"     // JNumber 123.0
run jNumber "-123"    // JNumber -123.0
run jNumber "123.4"   // JNumber 123.4
```

And what about some failing cases?

```
run jNumber "-123."   // JNumber -123.0 -- should fail!
run jNumber "00.1"    // JNumber 0      -- should fail!
```

Hmm. Something went wrong! These cases should fail, surely?

Well, no. What's happening in the `-123.` case is that the parser is consuming everything up the to decimal point and then stopping, leaving the decimal point to be matched by the next parser! So, not an error.

Similarly, in the `00.1` case, the parser is consuming only the first `0` then stopping, leaving the rest of the input (`0.4`) to be matched by the next parser. Again, not an error.

To fix this properly is out of scope, so let's just add some whitespace to the parser to force it to terminate.

```
let jNumber_ = jNumber .>> spaces1
```

Now let's test again:

```
run jNumber_ "123"     // JNumber 123.0
run jNumber_ "-123"    // JNumber -123.0

run jNumber_ "-123." |> printResult
// Line:0 Col:4 Error parsing number andThen many1 whitespace
// -123.
//     ^Unexpected '.'
```

and we find the error is being detected properly now.

Let's test the fractional part:

```
run jNumber_ "123.4"   // JNumber 123.4

run jNumber_ "00.4" |> printResult
// Line:0 Col:1 Error parsing number andThen many1 whitespace
// 00.4
//  ^Unexpected '0'
```

and the exponent part now:

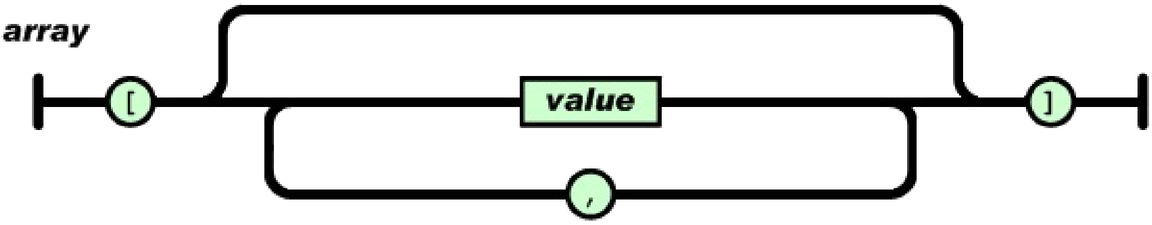
```
// exponent only
run jNumber_ "123e4"     // JNumber 1230000.0

// fraction and exponent
run jNumber_ "123.4e5"   // JNumber 12340000.0
run jNumber_ "123.4e-5"  // JNumber 0.001234
```

It's all looking good so far. Onwards and upwards!

5. Parsing `Array`

Next up is the `Array` case. Again, we can use the railway diagram to guide the implementation:



We will start with the primitives again. Note that we are adding optional whitespace after each token:

```
let jArray =

    let left = pchar '[' .>> spaces
    let right = pchar ']' .>> spaces
    let comma = pchar ',' .>> spaces
    let value = jValue .>> spaces
```

And then we create a list of values separated by a comma, with the whole list between the left and right brackets.

```
let jArray =
    ...

    // set up the list parser
    let values = sepBy1 value comma

    // set up the main parser
    between left values right
    |>> JArray
    <?> "array"
```

Hold on -- what is this `jValue` ?

```
let jArray =
    ...
    let value = jValue .>> spaces    // <=== what is "jValue"?
    ...
```

Well, the spec says that an `Array` can contain a list of values, so we'll assume that we have a `jValue` parser that can parse them.

But to parse a `JValue`, we need to parse a `Array` first!

We have hit a common problem in parsing -- mutually recursive definitions. We need a `JValue` parser to build an `Array`, but we need an `Array` parser to build a `JValue`.

How can we deal with this?

Forward references

The trick is to create a forward reference, a dummy `JValue` parser that we can use right now to define the `Array` parser, and then later on, we will fix up the forward reference with the "real" `JValue` parser.

This is one time where mutable references come in handy!

We will need a helper function to assist us with this, and the logic will be as follows:

- Define a dummy parser that will be replaced later.
- Define a real parser that forwards the input stream to the dummy parser.
- Return both the real parser and a reference to the dummy parser.

Now when the client fixes up the reference, the real parser will forward the input to the new parser that has replaced the dummy parser.

Here's the code:

```
let createParserForwardedToRef<'a>() =

    let dummyParser=
        let innerFn input : Result<'a * Input> = failwith "unfixed forwarded parser"
        {parseFn=innerFn; label="unknown"}

    // ref to placeholder Parser
    let parserRef = ref dummyParser

    // wrapper Parser
    let innerFn input =
        // forward input to the placeholder
        runOnInput !parserRef input
    let wrapperParser = {parseFn=innerFn; label="unknown"}

    wrapperParser, parserRef
```

With this in place, we can create a placeholder for a parser of type `JValue` :

```
let jValue,jValueRef = createParserForwardedToRef<JValue>()
```

Finishing up the `Array` parser

Going back to the `Array` parser, we can now compile it successfully, using the `jValue` placeholder:

```
let jArray =

    // set up the "primitive" parsers
    let left = pchar '[' .>> spaces
    let right = pchar ']' .>> spaces
    let comma = pchar ',' .>> spaces
    let value = jValue .>> spaces

    // set up the list parser
    let values = sepBy1 value comma

    // set up the main parser
    between left values right
    |>> JArray
    <?> "array"
```

If we try to test it now, we get an exception because we haven't fixed up the reference:

```
run jArray "[ 1, 2 ]"

// System.Exception: unfixed forwarded parser
```

So for now, let's fix up the reference to use one of the parsers that we have already created, such as `jNumber` :

```
jValueRef := jNumber
```

Now we *can* successfully test the `jArray` function, as long as we are careful to only use numbers in our array!

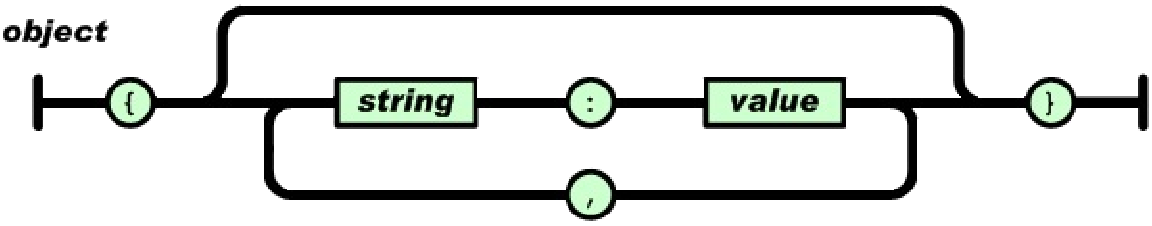
```
run jArray "[ 1, 2 ]"
// Success (JArray [JNumber 1.0; JNumber 2.0],

run jArray "[ 1, 2, ]" |> printResult
// Line:0 Col:6 Error parsing array
// [ 1, 2, ]
//       ^Unexpected ','
```

6. Parsing `Object`

The parser for `Object` is very similar to the one for `Array` .

First, the railway diagram:



Using this, we can create the parser directly, so I'll present it here without comment:

```
let jObject =

    // set up the "primitive" parsers
    let left = pchar '{' .>> spaces
    let right = pchar '}' .>> spaces
    let colon = pchar ':' .>> spaces
    let comma = pchar ',' .>> spaces
    let key = quotedString .>> spaces
    let value = jValue .>> spaces

    // set up the list parser
    let keyValue = (key .>> colon) .>>. value
    let keyValues = sepBy1 keyValue comma

    // set up the main parser
    between left keyValues right
    |>> Map.ofList  // convert the list of keyValues into a Map
    |>> JObject     // wrap in JObject
    <?> "object"    // add label
```

A bit of testing to make sure it works (but remember, only numbers are supported as values for now).

```
run jObject """{ "a":1, "b"  :  2 }"""
// JObject (map [("a", JNumber 1.0); ("b", JNumber 2.0)]),

run jObject """{ "a":1, "b"  :  2, }""" |> printResult
// Line:0 Col:18 Error parsing object
// { "a":1, "b"  :  2, }
//                   ^Unexpected ','
```

7. Putting it all together

Finally, we can combine all six of the parsers using the `choice` combinator, and we can assign this to the `JValue` parser reference that we created earlier:

```
jValueRef := choice
    [
    jNull
    jBool
    jNumber
    jString
    jArray
    jObject
    ]
```

And now we are ready to rock and roll!

Testing the complete parser: example 1

Here's an example of a JSON string that we can attempt to parse:

```
let example1 = """{
    "name" : "Scott",
    "isMale" : true,
    "bday" : {"year":2001, "month":12, "day":25 },
    "favouriteColors" : ["blue", "green"]
}"""
run jValue example1
```

And here is the result:

```
JObject
    (map
        [("bday", JObject(map
                [("day", JNumber 25.0);
                ("month", JNumber 12.0);
                ("year", JNumber 2001.0)]));
        ("favouriteColors", JArray [JString "blue"; JString "green"]);
        ("isMale", JBool true);
        ("name", JString "Scott")
        ])
```

Testing the complete parser: example 2

Here's one from the example page on json.org:

```
let example2= """{"widget": {
    "debug": "on",
    "window": {
        "title": "Sample Konfabulator Widget",
        "name": "main_window",
        "width": 500,
        "height": 500
    },
    "image": {
        "src": "Images/Sun.png",
        "name": "sun1",
        "hOffset": 250,
        "vOffset": 250,
        "alignment": "center"
    },
    "text": {
        "data": "Click Here",
        "size": 36,
        "style": "bold",
        "name": "text1",
        "hOffset": 250,
        "vOffset": 100,
        "alignment": "center",
        "onMouseUp": "sun1.opacity = (sun1.opacity / 100) * 90;"
    }
}}  """

run jValue example2
```

And here is the result:

```
JObject(map
    [("widget",JObject(map
            [("debug", JString "on");
            ("image",JObject(map
                [("alignment", JString "center");
                    ("hOffset", JNumber 250.0); ("name", JString "sun1");
                    ("src", JString "Images/Sun.png");
                    ("vOffset", JNumber 250.0)]));
            ("text",JObject(map
                [("alignment", JString "center");
                    ("data", JString "Click Here");
                    ("hOffset", JNumber 250.0);
                    ("name", JString "text1");
                    ("onMouseUp", JString "sun1.opacity = (sun1.opacity / 100) * 90;")
;
                    ("size", JNumber 36.0);
                    ("style", JString "bold");
                    ("vOffset", JNumber 100.0)]));
            ("window",JObject(map
                [("height", JNumber 500.0);
                    ("name", JString "main_window");
                    ("title", JString "Sample Konfabulator Widget");
                    ("width", JNumber 500.0)]))]))]),
```

Complete listing of the JSON parser

Here's the complete listing for the JSON parser -- it's about 250 lines of useful code.

The source code displayed below is also available at this gist.

```
#load "ParserLibrary.fsx"

open System
open ParserLibrary

(*
// --------------------------------
JSON spec from http://www.json.org/
// --------------------------------

The JSON spec is available at [json.org](http://www.json.org/). I'll paraphase it here
:

* A `value` can be a `string` or a `number` or a `bool` or `null` or an `object` or an
 `array`.
  * These structures can be nested.
* A `string` is a sequence of zero or more Unicode characters, wrapped in double quote
s, using backslash escapes.
* A `number` is very much like a C or Java number, except that the octal and hexadecim
```

```
al formats are not used.
* A `boolean` is the literal `true` or `false`
* A `null` is the literal `null`
* An `object` is an unordered set of name/value pairs.
  * An object begins with { (left brace) and ends with } (right brace).
  * Each name is followed by : (colon) and the name/value pairs are separated by , (co
mma).
* An `array` is an ordered collection of values.
  * An array begins with [ (left bracket) and ends with ] (right bracket).
  * Values are separated by , (comma).
* Whitespace can be inserted between any pair of tokens.

*)

type JValue =
    | JString of string
    | JNumber of float
    | JBool   of bool
    | JNull
    | JObject of Map<string, JValue>
    | JArray  of JValue list


// ======================================
// Forward reference
// ======================================

/// Create a forward reference
let createParserForwardedToRef<'a>() =

    let dummyParser=
        let innerFn input : Result<'a * Input> = failwith "unfixed forwarded parser"
        {parseFn=innerFn; label="unknown"}

    // ref to placeholder Parser
    let parserRef = ref dummyParser

    // wrapper Parser
    let innerFn input =
        // forward input to the placeholder
        runOnInput !parserRef input
    let wrapperParser = {parseFn=innerFn; label="unknown"}

    wrapperParser, parserRef

let jValue,jValueRef = createParserForwardedToRef<JValue>()

// ======================================
// Utility function
// ======================================

// applies the parser p, ignores the result, and returns x.
let (>>%) p x =
```

```
    p |>> (fun _ -> x)

// ======================================
// Parsing a JNull
// ======================================

let jNull =
    pstring "null"
    >>% JNull   // map to JNull
    <?> "null"  // give it a label

// ======================================
// Parsing a JBool
// ======================================

let jBool =
    let jtrue =
        pstring "true"
        >>% JBool true   // map to JBool
    let jfalse =
        pstring "false"
        >>% JBool false  // map to JBool

    // choose between true and false
    jtrue <|> jfalse
    <?> "bool"           // give it a label


// ======================================
// Parsing a JString
// ======================================

/// Parse an unescaped char
let jUnescapedChar =
    satisfy (fun ch -> ch <> '\\' && ch <> '\"') "char"

/// Parse an escaped char
let jEscapedChar =
    [
    // (stringToMatch, resultChar)
    ("\\\"",'\"')      // quote
    ("\\\\",'\\')      // reverse solidus
    ("\\/",'/')        // solidus
    ("\\b",'\b')       // backspace
    ("\\f",'\f')       // formfeed
    ("\\n",'\n')       // newline
    ("\\r",'\r')       // cr
    ("\\t",'\t')       // tab
    ]
    // convert each pair into a parser
    |> List.map (fun (toMatch,result) ->
        pstring toMatch >>% result)
    // and combine them into one
```

```
    |> choice

/// Parse a unicode char
let jUnicodeChar =

    // set up the "primitive" parsers
    let backslash = pchar '\\'
    let uChar = pchar 'u'
    let hexdigit = anyOf (['0'..'9'] @ ['A'..'F'] @ ['a'..'f'])

    // convert the parser output (nested tuples)
    // to a char
    let convertToChar (((h1,h2),h3),h4) =
        let str = sprintf "%c%c%c%c" h1 h2 h3 h4
        Int32.Parse(str,Globalization.NumberStyles.HexNumber) |> char

    // set up the main parser
    backslash  >>. uChar >>. hexdigit .>>. hexdigit .>>. hexdigit .>>. hexdigit
    |>> convertToChar


/// Parse a quoted string
let quotedString =
    let quote = pchar '\"' <?> "quote"
    let jchar = jUnescapedChar <|> jEscapedChar <|> jUnicodeChar

    // set up the main parser
    quote >>. manyChars jchar .>> quote

/// Parse a JString
let jString =
    // wrap the string in a JString
    quotedString
    |>> JString           // convert to JString
    <?> "quoted string"   // add label

// ======================================
// Parsing a JNumber
// ======================================

/// Parse a JNumber
let jNumber =

    // set up the "primitive" parsers
    let optSign = opt (pchar '-')

    let zero = pstring "0"

    let digitOneNine =
        satisfy (fun ch -> Char.IsDigit ch && ch <> '0') "1-9"

    let digit =
        satisfy (fun ch -> Char.IsDigit ch ) "digit"
```

```
    let point = pchar '.'

    let e = pchar 'e' <|> pchar 'E'

    let optPlusMinus = opt (pchar '-' <|> pchar '+')

    let nonZeroInt =
        digitOneNine .>>. manyChars digit
        |>> fun (first,rest) -> string first + rest

    let intPart = zero <|> nonZeroInt

    let fractionPart = point >>. manyChars1 digit

    let exponentPart = e >>. optPlusMinus .>>. manyChars1 digit

    // utility function to convert an optional value to a string, or "" if missing
    let ( |>? ) opt f =
        match opt with
        | None -> ""
        | Some x -> f x

    let convertToJNumber (((optSign,intPart),fractionPart),expPart) =
        // convert to strings and let .NET parse them! - crude but ok for now.

        let signStr =
            optSign
            |>? string   // e.g. "-"

        let fractionPartStr =
            fractionPart
            |>? (fun digits -> "." + digits )  // e.g. ".456"

        let expPartStr =
            expPart
            |>? fun (optSign, digits) ->
                let sign = optSign |>? string
                "e" + sign + digits          // e.g. "e-12"

        // add the parts together and convert to a float, then wrap in a JNumber
        (signStr + intPart + fractionPartStr + expPartStr)
        |> float
        |> JNumber

    // set up the main parser
    optSign .>>. intPart .>>. opt fractionPart .>>. opt exponentPart
    |>> convertToJNumber
    <?> "number"   // add label

// ======================================
// Parsing a JArray
// ======================================
```

```
let jArray =

    // set up the "primitive" parsers
    let left = pchar '[' .>> spaces
    let right = pchar ']' .>> spaces
    let comma = pchar ',' .>> spaces
    let value = jValue .>> spaces

    // set up the list parser
    let values = sepBy1 value comma

    // set up the main parser
    between left values right
    |>> JArray
    <?> "array"

// ======================================
// Parsing a JObject
// ======================================


let jObject =

    // set up the "primitive" parsers
    let left = pchar '{' .>> spaces
    let right = pchar '}' .>> spaces
    let colon = pchar ':' .>> spaces
    let comma = pchar ',' .>> spaces
    let key = quotedString .>> spaces
    let value = jValue .>> spaces

    // set up the list parser
    let keyValue = (key .>> colon) .>>. value
    let keyValues = sepBy1 keyValue comma

    // set up the main parser
    between left keyValues right
    |>> Map.ofList  // convert the list of keyValues into a Map
    |>> JObject     // wrap in JObject
    <?> "object"    // add label

// ======================================
// Fixing up the jValue ref
// ======================================

// fixup the forward ref
jValueRef := choice
    [
    jNull
    jBool
    jNumber
    jString
```

```
    jArray
    jObject
    ]
```

Summary

In this post, we built a JSON parser using the parser library that we have developed over the previous posts.

I hope that, by building both the parser library and a real-world parser from scratch, you have gained a good appreciation for how parser combinators work, and how useful they are.

I'll repeat what I said in the first post: if you are interesting in using this technique in production, be sure to investigate the FParsec library for F#, which is optimized for real-world usage.

And if you are using languages other than F#, there is almost certainly a parser combinator library available to use.

- For more information about parser combinators in general, search the internet for "Parsec", the Haskell library that influenced FParsec.
- For some more examples of using FParsec, try one of these posts:
  - Implementing a phrase search query for FogCreek's Kiln
  - A LOGO Parser
  - A Small Basic Parser
  - A C# Parser and building a C# compiler in F#
  - Write Yourself a Scheme in 48 Hours in F#
  - Parsing GLSL, the shading language of OpenGL

Thanks!

The source code for this post is available at this gist.

In this series of posts, I'll look at how you can thread state through a series of pure functions in a convenient way.

To start with, I'll tell the story of Dr Frankenfunctor and the Monadster, and how the Doctor needed a way to create "recipes" that were activated when lightning struck.

The Doctor then devised ways to work with these recipes using functions such as `map`, `bind` and `apply`.

In the final post, we'll see how we can use a computation expression to make the coding cleaner, and how these techniques can be generalized into the so-called "state monad".

Warning! These posts contains gruesome topics, strained analogies, discussion of monads

- Dr Frankenfunctor and the Monadster. Or, how a 19th century scientist nearly invented the state monad.
- Completing the body of the Monadster. Dr Frankenfunctor and the Monadster, part 2.
- Refactoring the Monadster. Dr Frankenfunctor and the Monadster, part 3.

Dr Frankenfunctor and the Monadster

UPDATE: Slides and video from my talk on this topic

Warning! This post contains gruesome topics, strained analogies, discussion of monads

For generations, we have been captivated by the tragic story of Dr Frankenfunctor. The fascination with vital forces, the early experiments with electricity and galvanism, and finally the breakthough culminating in the bringing to life of a collection of dead body parts -- the Monadster.

But then, as we all know, the creature escaped and the free Monadster rampaged through computer science conferences, bringing fear to the hearts of even the most seasoned programmers.



CAPTION: The terrible events at the 1990 ACM Conference on LISP and Functional Programming.

I will not repeat the details here; the story is still too terrible to recall.

But in all the millions of words devoted to this tragedy, one topic has never been satisfactorily addressed.

How was the creature assembled and brought to life?

We know that Dr Frankenfunctor built the creature from dead body parts, and then animated them in a single instant, using a bolt of lightning to create the vital force.

But the various body parts had to be assembled into a whole, and the vital force had to be transmitted through the assembly in the appropriate manner, and all this done in a split second, in the moment that the lightning struck.

I have devoted many years of research into this matter, and recently, at great expense, I have managed to obtain Dr Frankenfunctor's personal laboratory notebooks.

So at last, I can present Dr Frankenfunctor's technique to the world. Use it as you will. I do not make any judgements as to its morality, after all, it is not for mere developers to question the real-world effects of what we build.

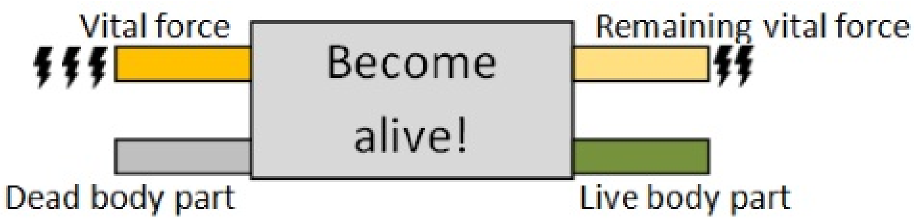
Background

To start with, you need to understand the fundamental process involved.

First, you must know that no whole body was available to Dr Frankenfunctor. Instead, the creature was created from an assemblage of body parts -- arms, legs, brain, heart -- whose provenances were murky and best left unspoken.

Dr Frankenfunctor started with a dead body part, and infused it with some amount of vital force. The result was two things: a now live body part, and the remaining, diminished, vital force, because of course some of the vital force was transferred to the live part.

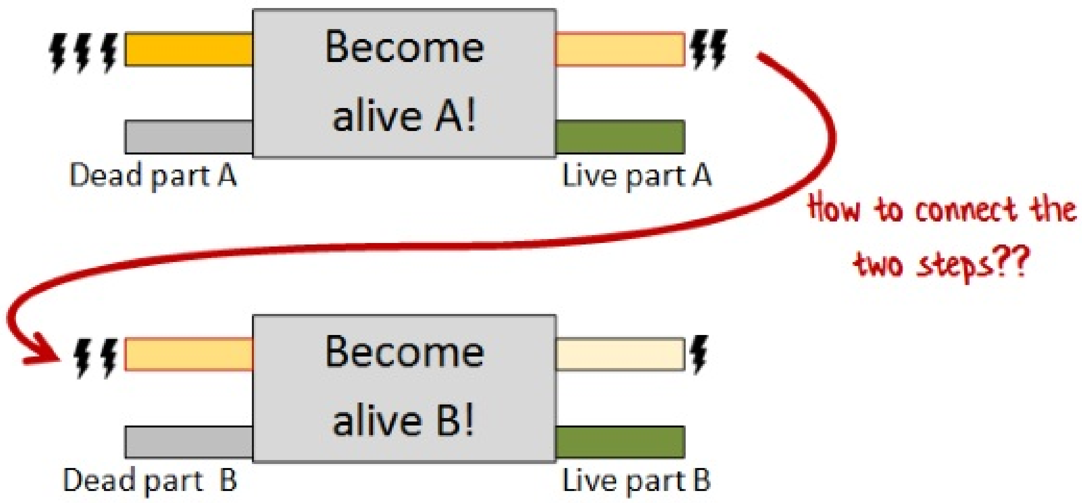
Here is a diagram demonstrating the principle:



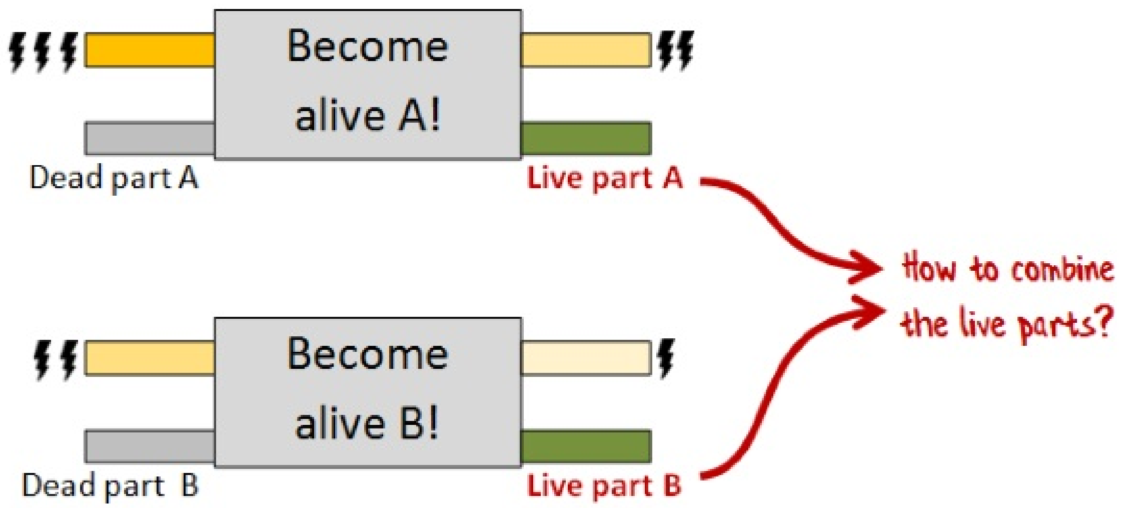
But this creates only *one* body part. How can we create more than one? This is the challenge that faced Dr Frankenfunctor.

The first problem is that we only have a limited quantity of the vital force. This means that when we need to animate a second body part, we have available only the remaining vital force from a previous step.

How can we connect the two steps together so that the vital force from the first step is fed into the input of the second step?



Even if we have chained the steps correctly, we need to take the various live body parts and combine them somehow. But we only have access to *live* body parts during the moment of creation. How can we combine them in that split second?



It was Dr Frankenfunctor's genius that led to an elegant approach that solved both of these problems, the approach that I will present to you now.

The common context

Before discussing the particulars of assembling the body parts, we should spend a moment on common functionality that is required for the rest of the procedure.

First, we need a label type. Dr Frankenfunctor was very disciplined in labeling the source of every part used.

```
type Label = string
```

The vital force we will model with a simple record type:

```
type VitalForce = {units:int}
```

Since we will be using vital force frequently, we will create a function that extracts one unit and returns a tuple of the unit and remaining force.

```
let getVitalForce vitalForce =
   let oneUnit = {units = 1}
   let remaining = {units = vitalForce.units-1}  // decrement
   oneUnit, remaining  // return both
```

The Left Leg

With the common code out of the way, we can return to the substance.

Dr Frankenfunctor's notebooks record that the lower extremities were created first. There was a left leg lying around in the laboratory, and that was the starting point.

```
type DeadLeftLeg = DeadLeftLeg of Label
```

From this leg, a live leg could be created with the same label and one unit of vital force.

```
type LiveLeftLeg = LiveLeftLeg of Label * VitalForce
```

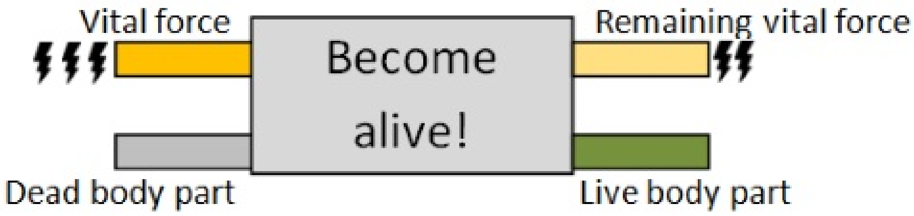
The type signature for the creation function would thus look like this:

```
type MakeLiveLeftLeg =
    DeadLeftLeg * VitalForce -> LiveLeftLeg * VitalForce
```

And the actual implementation like this:

```
let makeLiveLeftLeg (deadLeftLeg,vitalForce) =
    // get the label from the dead leg using pattern matching
    let (DeadLeftLeg label) = deadLeftLeg
    // get one unit of vital force
    let oneUnit, remainingVitalForce = getVitalForce vitalForce
    // create a live leg from the label and vital force
    let liveLeftLeg = LiveLeftLeg (label,oneUnit)
    // return the leg and the remaining vital force
    liveLeftLeg, remainingVitalForce
```

As you can see, this implementation matched the earlier diagram precisely.



At this point Dr Frankenfunctor had two important insights.

The first insight was that, thanks to currying, the function could be converted from a function taking a tuple to a two parameter function, with each parameter passed in turn.



And the code now looked like this:

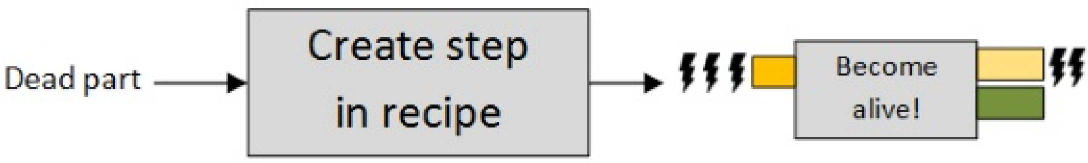
```
type MakeLiveLeftLeg =
    DeadLeftLeg -> VitalForce -> LiveLeftLeg * VitalForce

let makeLiveLeftLeg deadLeftLeg vitalForce =
    let (DeadLeftLeg label) = deadLeftLeg
    let oneUnit, remainingVitalForce = getVitalForce vitalForce
    let liveLeftLeg = LiveLeftLeg (label,oneUnit)
    liveLeftLeg, remainingVitalForce
```

The second insight was that this *same* code can be interpreted as a function that in turn returns a "becomeAlive" function.

That is, we have the dead part on hand, but we won't have any vital force until the final moment, so why not process the dead part right now and return a function that can be used when the vital force becomes available.

In other words, we pass in a dead part, and we get back a function that creates a live part when given some vital force.



These "become alive" functions can then be treated as "steps in a recipe", assuming we can find some way of combining them.

The code looks like this now:

```
type MakeLiveLeftLeg =
    DeadLeftLeg -> (VitalForce -> LiveLeftLeg * VitalForce)

let makeLiveLeftLeg deadLeftLeg =
    // create an inner intermediate function
    let becomeAlive vitalForce =
        let (DeadLeftLeg label) = deadLeftLeg
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveLeftLeg = LiveLeftLeg (label,oneUnit)
        liveLeftLeg, remainingVitalForce
    // return it
    becomeAlive
```

It may not be obvious, but this is *exactly the same code* as the previous version, just written slightly differently.

This curried function (with two parameters) can be interpreted as a normal two parameter function, or it can be interpreted as a *one parameter* function that returns *another* one parameter function.

If this is not clear, consider the much simpler example of a two parameter `add` function:

```
let add x y =
    x + y
```

Because F# curries functions by default, that implementation is exactly the same as this one:

```
let add x =
    fun y -> x + y
```

Which, if we define an intermediate function, is also exactly the same as this one:

```
let add x =
    let addX y = x + y
    addX // return the function
```

Creating the Monadster type

Looking ahead, we can see that we can use a similar approach for all the functions that create live body parts.

All those functions will return a function that has a signature like: `VitalForce -> LiveBodyPart * VitalForce`.

To make our life easy, let's give that function signature a name, `M`, which stands for "Monadster part generator", and give it a generic type parameter `'LiveBodyPart` so that we can use it with many different body parts.

```
type M<'LiveBodyPart> =
    VitalForce -> 'LiveBodyPart * VitalForce
```

We can now explicitly annotate the return type of the `makeLiveLeftLeg` function with `:M<LiveLeftLeg>`.

```
let makeLiveLeftLeg deadLeftLeg :M<LiveLeftLeg> =
    let becomeAlive vitalForce =
        let (DeadLeftLeg label) = deadLeftLeg
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveLeftLeg = LiveLeftLeg (label,oneUnit)
        liveLeftLeg, remainingVitalForce
    becomeAlive
```

The rest of the function is unchanged because the `becomeAlive` return value is already compatible with `M<LiveLeftLeg>`.

But I don't like having to explicitly annotate all the time. How about we wrap the function in a single case union -- call it "M" -- to give it its own distinct type? Like this:

```
type M<'LiveBodyPart> =
    M of (VitalForce -> 'LiveBodyPart * VitalForce)
```

That way, we can distinguish between a "Monadster part generator" and an ordinary function returning a tuple.

To use this new definition, we need to tweak the code to wrap the intermediate function in the single case union `M` when we return it, like this:

```
let makeLiveLeftLegM deadLeftLeg  =
    let becomeAlive vitalForce =
        let (DeadLeftLeg label) = deadLeftLeg
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveLeftLeg = LiveLeftLeg (label,oneUnit)
        liveLeftLeg, remainingVitalForce
    // changed!
    M becomeAlive // wrap the function in a single case union
```

For this last version, the type signature will be correctly inferred without having to specify it explicitly: a function that takes a dead left leg and returns an "M" of a live leg:

```
val makeLiveLeftLegM : DeadLeftLeg -> M<LiveLeftLeg>
```

Note that I've renamed the function `makeLiveLeftLegM` to make it clear that it returns a `M` of `LiveLeftLeg`.

The meaning of M

So what does this "M" type mean exactly? How can we make sense of it?

One helpful way is to think of a `M<T>` as a *recipe* for creating a `T`. You give me some vital force and I'll give you back a `T`.

But how can an `M<T>` create a `T` out of nothing?

That's where functions like `makeLiveLeftLegM` are critically important. They take a parameter and "bake" it into the result. As a result, you will see lots of "M-making" functions with similar signatures, all looking something like this:



Or in code terms:

```
DeadPart -> M<LivePart>
```

The challenge now will be how to combine these in an elegant way.

Testing the left leg

Ok, let's test what we've got so far.

We'll start by creating a dead leg and use `makeLiveLeftLegM` on it to get an `M<LiveLeftLeg>`.

```
let deadLeftLeg = DeadLeftLeg "Boris"
let leftLegM = makeLiveLeftLegM deadLeftLeg
```

What is `leftLegM`? It's a recipe for creating a live left leg, given some vital force.

What's useful is that we can create this recipe *up front*, *before* the lightning strikes.

Now let's pretend that the storm has arrived, the lightning has struck, and 10 units of vital force are now available:

```
let vf = {units = 10}
```

Now, inside the `leftLegM` is a function which we can apply to the vital force. But first we need to get the function out of the wrapper using pattern matching.

```
let (M innerFn) = leftLegM
```

And then we can run the inner function to get the live left leg and the remaining vital force:

```
let liveLeftLeg, remainingAfterLeftLeg = innerFn vf
```

The results look like this:

```
val liveLeftLeg : LiveLeftLeg =
   LiveLeftLeg ("Boris",{units = 1;})
val remainingAfterLeftLeg : VitalForce =
   {units = 9;}
```

You can see that a `LiveLeftLeg` was created successfully and that the remaining vital force is reduced to 9 units now.

This pattern matching is awkward, so let's create a helper function that both unwraps the inner function and calls it, all in one go.

We'll call it `runM` and it looks like this:

```
let runM (M f) vitalForce = f vitalForce
```

So the test code above would now be simplified to this:

```
let liveLeftLeg, remainingAfterLeftLeg = runM leftLegM vf
```

So now, finally, we have a function that can create a live left leg.

It took a while to get it working, but we've also built some useful tools and concepts that we can use moving forwards.

The Right Leg

Now that we know what we are doing, we should be able to use the same techniques for the other body parts now.

How about a right leg then?

Unfortunately, according to the notebook, Dr Frankenfunctor could not find a right leg in the laboratory. The problem was solved with a hack... but we'll come to that later.

The Left Arm

Next, the arms were created, starting with the left arm.

But there was a problem. The laboratory only had a *broken* left arm lying around. The arm had to be healed before it could be used in the final body.

Now Dr Frankenfunctor, being a doctor, *did* know how to heal a broken arm, but only a live one. Trying to heal a dead broken arm would be impossible.

In code terms, we have this:

```
type DeadLeftBrokenArm = DeadLeftBrokenArm of Label

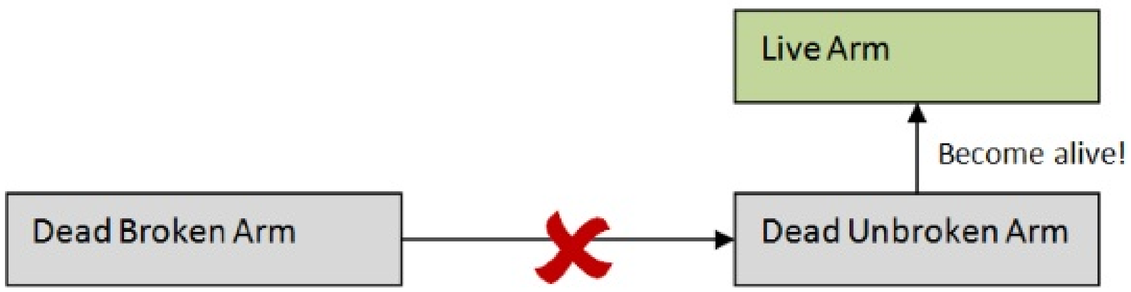
// A live version of the broken arm.
type LiveLeftBrokenArm = LiveLeftBrokenArm of Label * VitalForce

// A live version of a heathly arm, with no dead version available
type LiveLeftArm = LiveLeftArm of Label * VitalForce

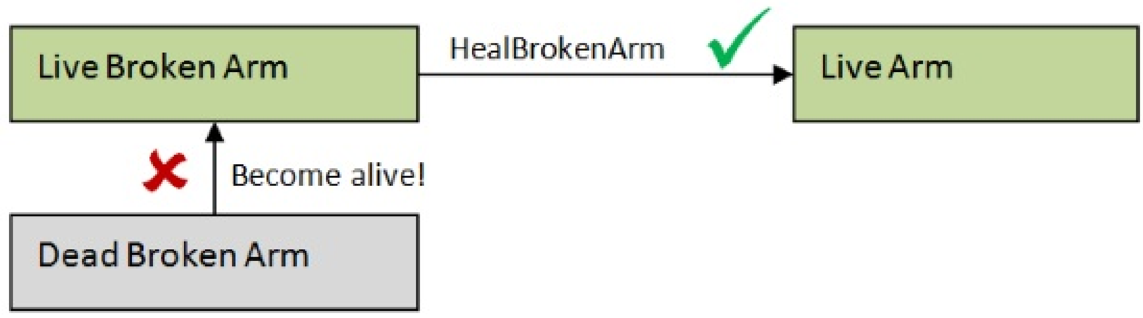
// An operation that can turn a broken left arm into a heathly left arm
type HealBrokenArm = LiveLeftBrokenArm -> LiveLeftArm
```

The challenge was therefore this: how can we make a live left arm out the material we have on hand?

First, we have to rule out creating a `LiveLeftArm` from a `DeadLeftUnbrokenArm`, as there isn't any such thing. Nor can we convert a `DeadLeftBrokenArm` into a healthy `LiveLeftArm` directly.

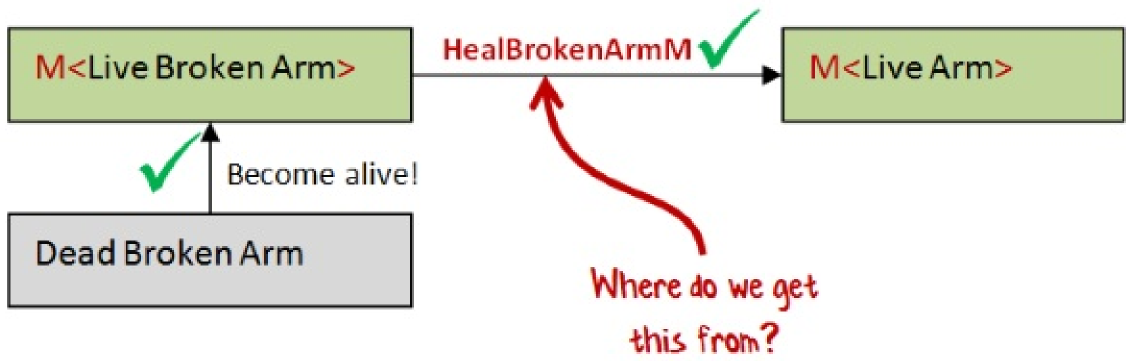


But what we *can* do is turn the `DeadLeftBrokenArm` into a *live* broken arm and then heal the live broken arm, yes?



No, I'm afraid that won't work. We can't create live parts directly, we can only create live parts in the context of the `M` recipe.

What we need to do then is create a special version of `healBrokenArm` (call it `healBrokenArmM`) that converts a `M<LiveBrokenArm>` to a `M<LiveArm>` .



But how do we create such a function? And how can we reuse `healBrokenArm` as part of it?

Let's start with the most straightforward implementation.

First, since the function will return an `M` something, it will have the same form as the `makeLiveLeftLegM` function that we saw earlier. We'll need to create an inner function that has a vitalForce parameter, and then return it wrapped in an `M` .

But unlike the function that we saw earlier, this one has an `M` as parameter too (an `M<LiveBrokenArm>`). How can we extract the data we need from this input?

Simple, just run it with some vitalForce. And where are we going to get the vitalForce from? From the parameter to the inner function!

So our finished version will look like this:

```
// implementation of HealBrokenArm
let healBrokenArm (LiveLeftBrokenArm (label,vf)) = LiveLeftArm (label,vf)

/// convert a M<LiveLeftBrokenArm> into a M<LiveLeftArm>
let makeHealedLeftArm brokenArmM =

    // create a new inner function that takes a vitalForce parameter
    let healWhileAlive vitalForce =
        // run the incoming brokenArmM with the vitalForce
        // to get a broken arm
        let brokenArm,remainingVitalForce = runM brokenArmM vitalForce

        // heal the broken arm
        let healedArm = healBrokenArm brokenArm

        // return the healed arm and the remaining VitalForce
        healedArm, remainingVitalForce

    // wrap the inner function and return it
    M healWhileAlive
```

If we evaluate this code, we get the signature:

```
val makeHealedLeftArm : M<LiveLeftBrokenArm> -> M<LiveLeftArm>
```

which is exactly what we want!

But not so fast -- we can do better.

We've hard-coded the `healBrokenArm` transformation in there. What happens if we want to do some other transformation, and for some other body part? Can we make this function a bit more generic?

Yes, it's easy. All we need to is pass in a function ("f" say) that transforms the body part, like this:

```
let makeGenericTransform f brokenArmM =

    // create a new inner function that takes a vitalForce parameter
    let healWhileAlive vitalForce =
        let brokenArm,remainingVitalForce = runM brokenArmM vitalForce

        // heal the broken arm using passed in f
        let healedArm = f brokenArm
        healedArm, remainingVitalForce

    M healWhileAlive
```

What's amazing about this is that by parameterizing that one transformation with the `f` parameter, the *whole* function becomes generic!

We haven't made any other changes, but the signature for `makeGenericTransform` no longer refers to arms. It works with anything!

```
val makeGenericTransform : f:('a -> 'b) -> M<'a> -> M<'b>
```

Introducing mapM

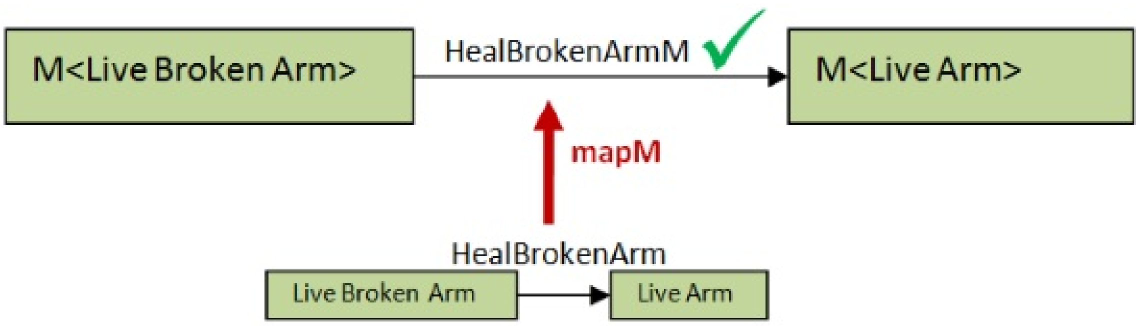
Since it is so generic now, the names are confusing. Let's rename it. I'll call it `mapM`. It works with *any* body part and *any* transformation.

Here's the implementation, with the internal names fixed up too.

```
let mapM f bodyPartM =
    let transformWhileAlive vitalForce =
        let bodyPart,remainingVitalForce = runM bodyPartM vitalForce
        let updatedBodyPart = f bodyPart
        updatedBodyPart, remainingVitalForce
    M transformWhileAlive
```

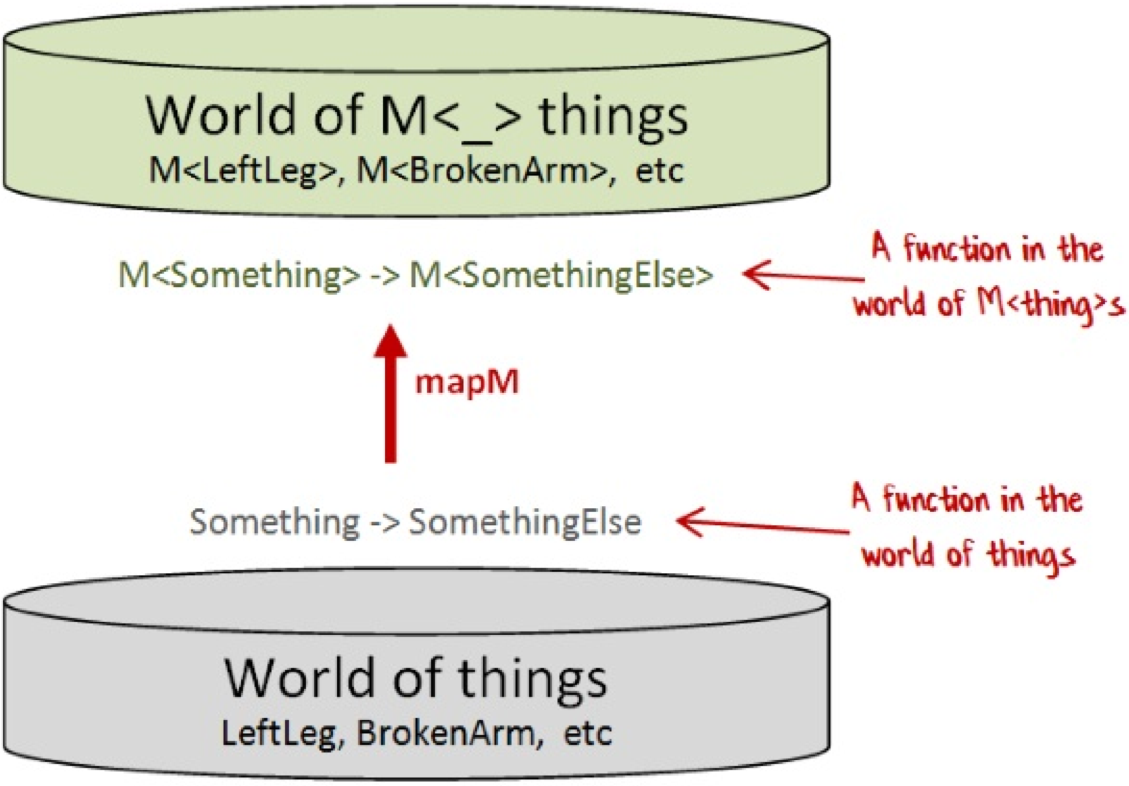
In particular, it works with the `healBrokenArm` function, so to create a version of "heal" that has been lifted to work with `M` s we can just write this:

```
let healBrokenArmM = mapM healBrokenArm
```



The importance of mapM

One way of thinking about `mapM` is that it is a "function converter". Given any "normal" function, it converts it to a function where the input and output are `M` s.



Functions similar to `mapM` crop up in many situations. For example, `Option.map` transforms a "normal" function into a function whose inputs and outputs are options. Similarly, `List.map` transforms a "normal" function into a function whose inputs and outputs are lists. And there are many other examples.

```
// map works with options
let healBrokenArmO = Option.map healBrokenArm
// LiveLeftBrokenArm option -> LiveLeftArm option

// map works with lists
let healBrokenArmL = List.map healBrokenArm
// LiveLeftBrokenArm list -> LiveLeftArm list
```

What might be new to you is that the "wrapper" type `M` contains a *function*, not a simple data structure like Option or List. That might make your head hurt!

In addition, the diagram above implies that `M` could wrap *any* normal type and `mapM` could map *any* normal function.

Let's try it and see!

```
let isEven x = (x%2 = 0)   // int -> bool
// map it
let isEvenM = mapM isEven  // M<int> -> M<bool>

let isEmpty x = (String.length x)=0  // string -> bool
// map it
let isEmptyM = mapM isEmpty          // M<string> -> M<bool>
```

So, yes, it works!

Testing the left arm

Again, let's test what we've got so far.

We'll start by creating a dead broken arm and use `makeLiveLeftBrokenArm` on it to get an `M<BrokenLeftArm>`.

```
let makeLiveLeftBrokenArm deadLeftBrokenArm =
    let (DeadLeftBrokenArm label) = deadLeftBrokenArm
    let becomeAlive vitalForce =
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveLeftBrokenArm = LiveLeftBrokenArm (label,oneUnit)
        liveLeftBrokenArm, remainingVitalForce
    M becomeAlive

/// create a dead Left Broken Arm
let deadLeftBrokenArm = DeadLeftBrokenArm "Victor"

/// create a M<BrokenLeftArm> from the dead one
let leftBrokenArmM = makeLiveLeftBrokenArm deadLeftBrokenArm
```

Now we can use `mapM` and `healBrokenArm` to convert the `M<BrokenLeftArm>` into a `M<LeftArm>`:

```
let leftArmM = leftBrokenArmM |> mapM healBrokenArm
```

What we have now in `leftArmM` is a recipe for creating a unbroken and live left arm. All we need to do is add some vital force.

As before, we can do all these things up front, before the lightning strikes.

Now when the storm arrives, and the lightning has struck, and vital force is available, we can run `leftArmM` with the vital force...

```
let vf = {units = 10}

let liveLeftArm, remainingAfterLeftArm = runM leftArmM vf
```

...and we get this result:

```
val liveLeftArm : LiveLeftArm =
    LiveLeftArm ("Victor",{units = 1;})
val remainingAfterLeftArm :
    VitalForce = {units = 9;}
```

A live left arm, just as we wanted.

The Right Arm

On to the right arm next.

Again, there was a problem. Dr Frankenfunctor's notebooks record that there was no whole arm available. However there *was* a lower arm and an upper arm...

```
type DeadRightLowerArm = DeadRightLowerArm of Label
type DeadRightUpperArm = DeadRightUpperArm of Label
```

...which could be turned into corresponding live ones:

```
type LiveRightLowerArm = LiveRightLowerArm of Label * VitalForce
type LiveRightUpperArm = LiveRightUpperArm of Label * VitalForce
```

Dr Frankenfunctor decided to do surgery to join the two arm sections into a whole arm.

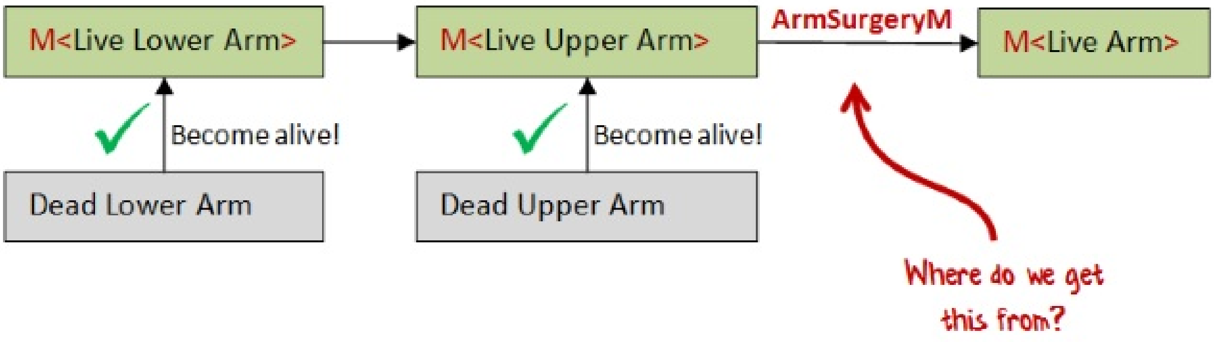
```
// define the whole arm
type LiveRightArm = {
    lowerArm : LiveRightLowerArm
    upperArm : LiveRightUpperArm
    }

// surgery to combine the two arm parts
let armSurgery lowerArm upperArm =
    {lowerArm=lowerArm; upperArm=upperArm}
```

As with the broken arm, the surgery could only be done with *live* parts. Doing that with dead parts would be yucky and gross.

But also, as with the broken arm, we don't have access to the live parts directly, only within the context of an `M` wrapper.

In other words we need to convert our `armSurgery` function that works with normal live parts, and convert it into a `armSurgeryM` function that works with `M` s.



We can use the same approach as we did before:

- create a inner function that takes a vitalForce parameter
- run the incoming parameters with the vitalForce to extract the data
- from the inner function return the new data after surgery
- wrap the inner function in an "M" and return it

Here's the code:

```
/// convert a M<LiveRightLowerArm> and  M<LiveRightUpperArm> into a M<LiveRightArm>
let makeArmSurgeryM_v1 lowerArmM upperArmM =

    // create a new inner function that takes a vitalForce parameter
    let becomeAlive vitalForce =
        // run the incoming lowerArmM with the vitalForce
        // to get the lower arm
        let liveLowerArm,remainingVitalForce = runM lowerArmM vitalForce

        // run the incoming upperArmM with the remainingVitalForce
        // to get the upper arm
        let liveUpperArm,remainingVitalForce2 = runM upperArmM remainingVitalForce

        // do the surgery to create a liveRightArm
        let liveRightArm = armSurgery liveLowerArm liveUpperArm

        // return the whole arm and the SECOND remaining VitalForce
        liveRightArm, remainingVitalForce2

    // wrap the inner function and return it
    M becomeAlive
```

One big difference from the broken arm example is that we have *two* parameters, of course. When we run the second parameter (to get the `liveUpperArm`), we must be sure to pass in the *remaining vital force* after the first step, not the original one.

And then, when we return from the inner function, we must be sure to return `remainingVitalForce2` (the remainder after the second step) not any other one.

If we compile this code, we get:

```
M<LiveRightLowerArm> -> M<LiveRightUpperArm> -> M<LiveRightArm>
```

which is just the signature we are looking for.

Introducing map2M

But as before, why not make this more generic? We don't need to hard-code `armSurgery` -- we can pass it as a parameter.

We'll call the more generic function `map2M` -- just like `mapM` but with two parameters.

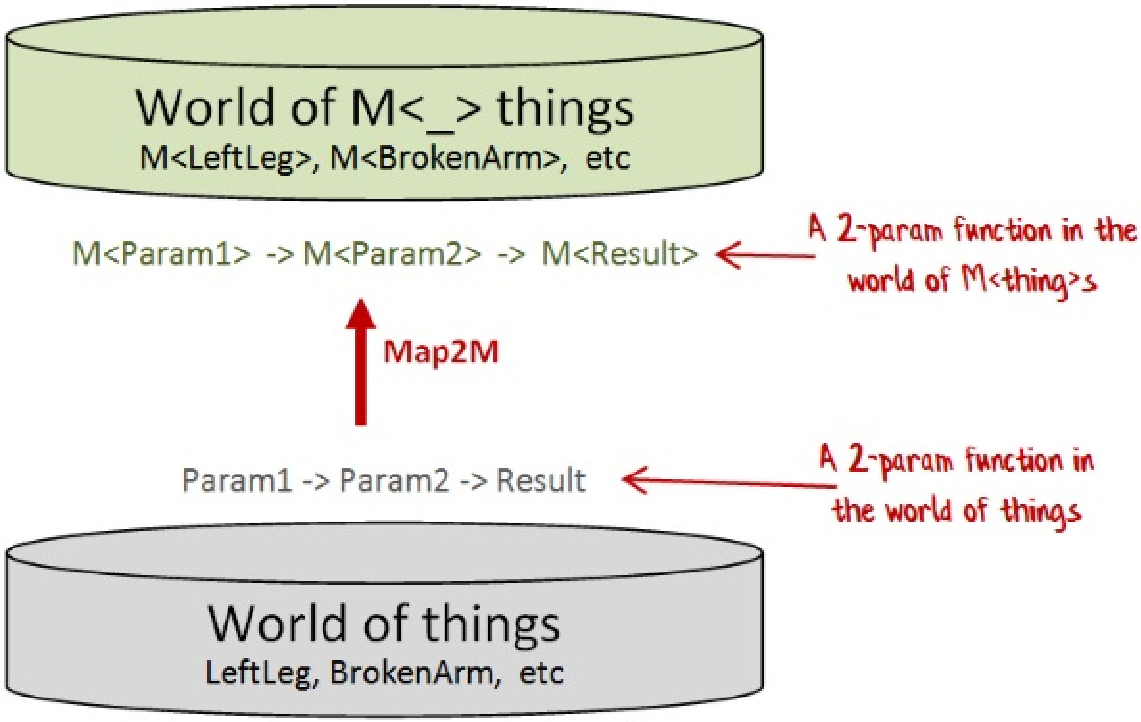
Here's the implementation:

```
let map2M f m1 m2 =
    let becomeAlive vitalForce =
        let v1,remainingVitalForce = runM m1 vitalForce
        let v2,remainingVitalForce2 = runM m2 remainingVitalForce
        let v3 = f v1 v2
        v3, remainingVitalForce2
    M becomeAlive
```

And it has the signature:

```
f:('a -> 'b -> 'c) -> M<'a> -> M<'b> -> M<'c>
```

Just as with `mapM` we can interpret this function as a "function converter" that converts a "normal" two parameter function into a function in the world of `M`.



Testing the right arm

Again, let's test what we've got so far.

As always, we need some functions to convert the dead parts into live parts.

```
let makeLiveRightLowerArm (DeadRightLowerArm label) =
    let becomeAlive vitalForce =
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveRightLowerArm = LiveRightLowerArm (label,oneUnit)
        liveRightLowerArm, remainingVitalForce
    M becomeAlive

let makeLiveRightUpperArm (DeadRightUpperArm label) =
    let becomeAlive vitalForce =
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveRightUpperArm = LiveRightUpperArm (label,oneUnit)
        liveRightUpperArm, remainingVitalForce
    M becomeAlive
```

By the way, are you noticing that there is a lot of duplication in these functions? Me too! We will attempt to fix that later.

Next, we'll create the parts:

```
let deadRightLowerArm = DeadRightLowerArm "Tom"
let lowerRightArmM = makeLiveRightLowerArm deadRightLowerArm

let deadRightUpperArm = DeadRightUpperArm "Jerry"
let upperRightArmM = makeLiveRightUpperArm deadRightUpperArm
```

And then create the function to make a whole arm:

```
let armSurgeryM  = map2M armSurgery
let rightArmM = armSurgeryM lowerRightArmM upperRightArmM
```

As always, we can do all these things up front, before the lightning strikes, building a recipe (or *computation* if you like) that will do everything we need when the time comes.

When the vital force is available, we can run `rightArmM` with the vital force...

```
let vf = {units = 10}

let liveRightArm, remainingFromRightArm = runM rightArmM vf
```

...and we get this result:

```
val liveRightArm : LiveRightArm =
    {lowerArm = LiveRightLowerArm ("Tom",{units = 1;});
     upperArm = LiveRightUpperArm ("Jerry",{units = 1;});}

val remainingFromRightArm : VitalForce =
    {units = 8;}
```

A live right arm, composed of two subcomponents, just as required.

Also note that the remaining vital force has gone down to *eight*. We have correctly used up two units of vital force.

Summary

In this post, we saw how to create a `M` type that wrapped a "become alive" function that in turn could only be activated when lightning struck.

We also saw how various M-values could be processed and combined using `mapM` (for the broken arm) and `map2M` (for the arm in two parts).

The code samples used in this post are available on GitHub.

Next time

This exciting tale has more shocks in store for you! Stay tuned for the next installment, when I reveal how the head and body were created.

Completing the body of the Monadster

UPDATE: Slides and video from my talk on this topic

Warning! This post contains gruesome topics, strained analogies, discussion of monads

Welcome to the gripping tale of Dr Frankenfunctor and the Monadster!

We saw in the previous installment how Dr Frankenfunctor created life out of dead body parts using "Monadster part generators" (or "M"s for short), that would, on being supplied with some vital force, return a live body part.

We also saw how the leg and arms of the creature were created, and how these M-values could be processed and combined using `mapM` (for the broken arm) and `map2M` (for the arm in two parts).

In this second installment, we'll look at the other techniques Dr Frankenfunctor used to create the head, the heart, and the complete body.

The Head

First, the head.

Just like the right arm, the head is composed of two parts, a brain and a skull.

Dr Frankenfunctor started by defining the dead brain and skull:

```
type DeadBrain = DeadBrain of Label
type Skull = Skull of Label
```

Unlike the two-part right arm, only the brain needs to become alive. The skull can be used as is and does not need to be transformed before being used in a live head.

```
type LiveBrain = LiveBrain of Label * VitalForce

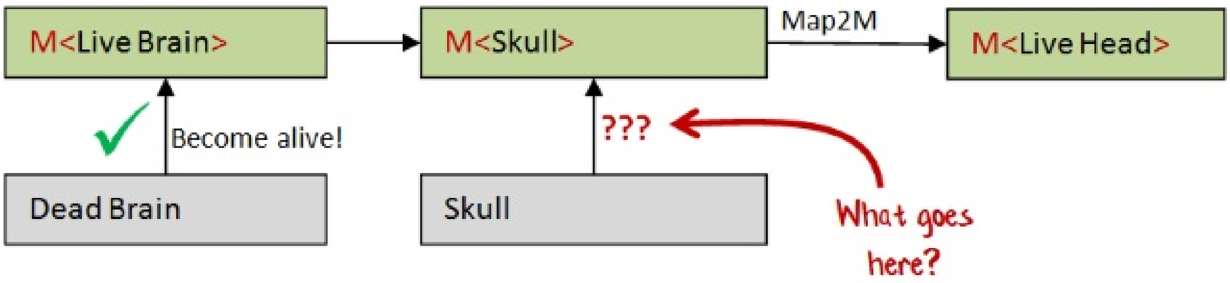
type LiveHead = {
    brain : LiveBrain
    skull : Skull // not live
    }
```

The live brain is combined with the skull to make a live head using a `headSurgery` function, analogous to the `armSurgery` we had earlier.

```
let headSurgery brain skull =
    {brain=brain; skull=skull}
```

Now we are ready to create a live head -- but how should we do it?

It would be great if we could reuse `map2M`, but there's a catch -- for `map2M` to work, it needs a skull wrapped in a `M`.



But the skull doesn't need to become alive or use vital force, so we will need to create a special function that converts a `Skull` to a `M<Skull>`.

We can use the same approach as we did before:

- create a inner function that takes a vitalForce parameter
- in this case, we leave the vitalForce untouched
- from the inner function return the original skull and the untouched vitalForce
- wrap the inner function in an "M" and return it

Here's the code:

```
let wrapSkullInM skull =
    let becomeAlive vitalForce =
        skull, vitalForce
    M becomeAlive
```

But the signature of `wrapSkullInM` is quite interesting.

```
val wrapSkullInM : 'a -> M<'a>
```

No mention of skulls anywhere!

Introducing returnM

We've created a completely generic function that will turn anything into an `M`. So let's rename it. I'm going to call it `returnM`, but in other contexts it might be called `pure` or `unit`.

```
let returnM x =
    let becomeAlive vitalForce =
        x, vitalForce
    M becomeAlive
```

Testing the head

Let's put this into action.

First, we need to define how to create a live brain.

```
let makeLiveBrain (DeadBrain label) =
    let becomeAlive vitalForce =
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveBrain = LiveBrain (label,oneUnit)
        liveBrain, remainingVitalForce
    M becomeAlive
```

Next we obtain a dead brain and skull:

```
let deadBrain = DeadBrain "Abby Normal"
let skull = Skull "Yorick"
```

By the way, how this particular dead brain was obtained is an interesting story that I don't have time to go into right now.



Next we build the "M" versions from the dead parts:

```
let liveBrainM = makeLiveBrain deadBrain
let skullM = returnM skull
```

And combine the parts using `map2M` :

```
let headSurgeryM = map2M headSurgery
let headM = headSurgeryM liveBrainM skullM
```

Once again, we can do all these things up front, before the lightning strikes.

When the vital force is available, we can run `headM` with the vital force...

```
let vf = {units = 10}

let liveHead, remainingFromHead = runM headM vf
```

...and we get this result:

```
val liveHead : LiveHead =
    {brain = LiveBrain ("Abby normal",{units = 1;});
    skull = Skull "Yorick";}

val remainingFromHead : VitalForce =
    {units = 9;}
```

A live head, composed of two subcomponents, just as required.

Also note that the remaining vital force is just nine, as the skull did not use up any units.

The Beating Heart

There is one more component we need, and that is a heart.

First, we have a dead heart and a live heart defined in the usual way:

```
type DeadHeart = DeadHeart of Label
type LiveHeart = LiveHeart of Label * VitalForce
```

But the creature needs more than a live heart -- it needs a *beating heart*. A beating heart is constructed from a live heart and some more vital force, like this:

```
type BeatingHeart = BeatingHeart of LiveHeart * VitalForce
```

The code that creates a live heart is very similar to the previous examples:

```
let makeLiveHeart (DeadHeart label) =
    let becomeAlive vitalForce =
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveHeart = LiveHeart (label,oneUnit)
        liveHeart, remainingVitalForce
    M becomeAlive
```

The code that creates a beating heart is also very similar. It takes a live heart as a parameter, uses up another unit of vital force, and returns the beating heart and the remaining vital force.

```
let makeBeatingHeart liveHeart =

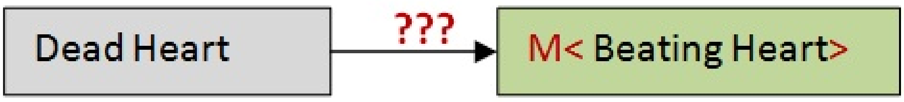
    let becomeAlive vitalForce =
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let beatingHeart = BeatingHeart (liveHeart, oneUnit)
        beatingHeart, remainingVitalForce
    M becomeAlive
```

If we look at the signatures for these functions, we see that they are very similar; both of the form `Something -> M<SomethingElse>`.

```
val makeLiveHeart : DeadHeart -> M<LiveHeart>
val makeBeatingHeart : LiveHeart -> M<BeatingHeart>
```

Chaining together M-returning functions

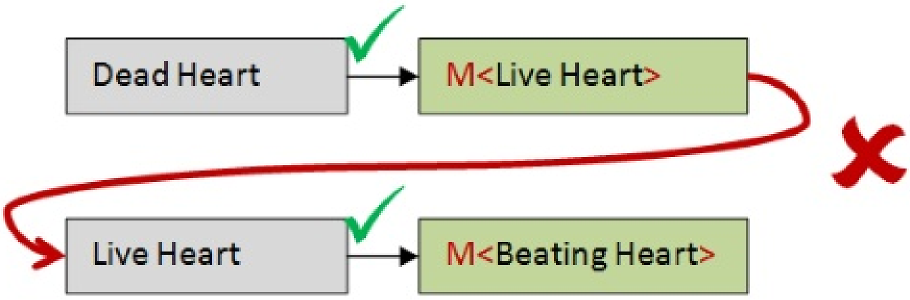
We start with a dead heart, and we need to get a beating heat



But we don't have the tools to do this directly.

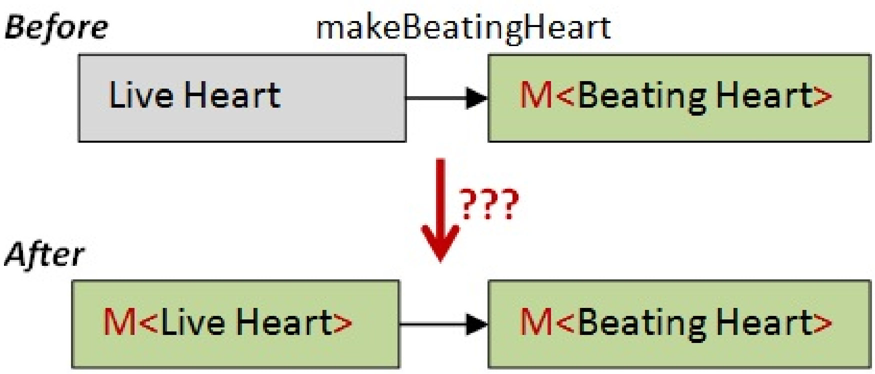
We have a function that turns a `DeadHeart` into a `M<LiveHeart>`, and we have a function that turns a `LiveHeart` into a `M<BeatingHeart>`.

But the output of the first is not compatible with the input of the second, so we can't glue them together.



What we want then, is a function that, given a `M<LiveHeart>` as input, can convert it to a `M<BeatingHeart>`.

And furthermore, we want to build it from the `makeBeatingHeart` function we already have.



Here's a first attempt, using the same pattern we've used many times before:

```
let makeBeatingHeartFromLiveHeartM liveHeartM =

    let becomeAlive vitalForce =
        // extract the liveHeart from liveHeartM
        let liveHeart, remainingVitalForce = runM liveHeartM vitalForce

        // use the liveHeart to create a beatingHeartM
        let beatingHeartM = makeBeatingHeart liveHeart

        // what goes here?

        // return a beatingHeart and remaining vital force
        beatingHeart, remainingVitalForce

    M becomeAlive
```

But what goes in the middle? How can we get a beating heart from a `beatingHeartM`? The answer is to run it with some vital force (which we happen to have on hand, because we are in the middle of the `becomeAlive` function).

What vital force though? It should be the remaining vital force after getting the `liveHeart`.

So the final version looks like this:

```
let makeBeatingHeartFromLiveHeartM liveHeartM =

    let becomeAlive vitalForce =
        // extract the liveHeart from liveHeartM
        let liveHeart, remainingVitalForce = runM liveHeartM vitalForce

        // use the liveHeart to create a beatingHeartM
        let beatingHeartM = makeBeatingHeart liveHeart

        // run beatingHeartM to get a beatingHeart
        let beatingHeart, remainingVitalForce2 = runM beatingHeartM remainingVitalForce

        // return a beatingHeart and remaining vital force
        beatingHeart, remainingVitalForce2

    // wrap the inner function and return it
    M becomeAlive
```

Notice that we return `remainingVitalForce2` at the end, the remainder after both steps are run.

If we look at the signature for this function, it is:

```
M<LiveHeart> -> M<BeatingHeart>
```

which is just what we wanted!

Introducing bindM

Once again, we can make this function generic by passing in a function parameter rather than hardcoding `makeBeatingHeart`.

I'll call it `bindM`. Here's the code:

```
let bindM f bodyPartM =
    let becomeAlive vitalForce =
        let bodyPart, remainingVitalForce = runM bodyPartM vitalForce
        let newBodyPartM = f bodyPart
        let newBodyPart, remainingVitalForce2 = runM newBodyPartM remainingVitalForce
        newBodyPart, remainingVitalForce2
    M becomeAlive
```

and the signature is:

```
f:('a -> M<'b>) -> M<'a> -> M<'b>
```

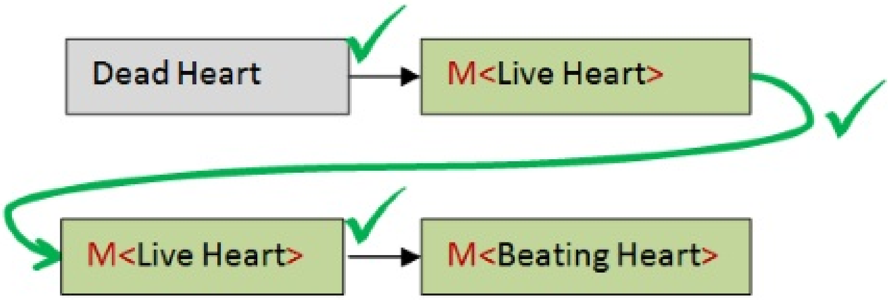
In other words, given any function `Something -> M<SomethingElse>` , I can convert it to a function `M<Something> -> M<SomethingElse>` that has an `M` as input and output.

By the way, functions with a signature like `Something -> M<SomethingElse>` are often called *monadic* functions.

Anyway, once you understand what is going on in `bindM` , a slightly shorter version can be implemented like this:

```
let bindM f bodyPartM =
    let becomeAlive vitalForce =
        let bodyPart, remainingVitalForce = runM bodyPartM vitalForce
        runM (f bodyPart) remainingVitalForce
    M becomeAlive
```

So finally, we have a way of creating a function, that given a `DeadHeart` , creates a `M<BeatingHeart>` .



Here's the code:

```
// create a dead heart
let deadHeart = DeadHeart "Anne"

// create a live heart generator (M<LiveHeart>)
let liveHeartM = makeLiveHeart deadHeart

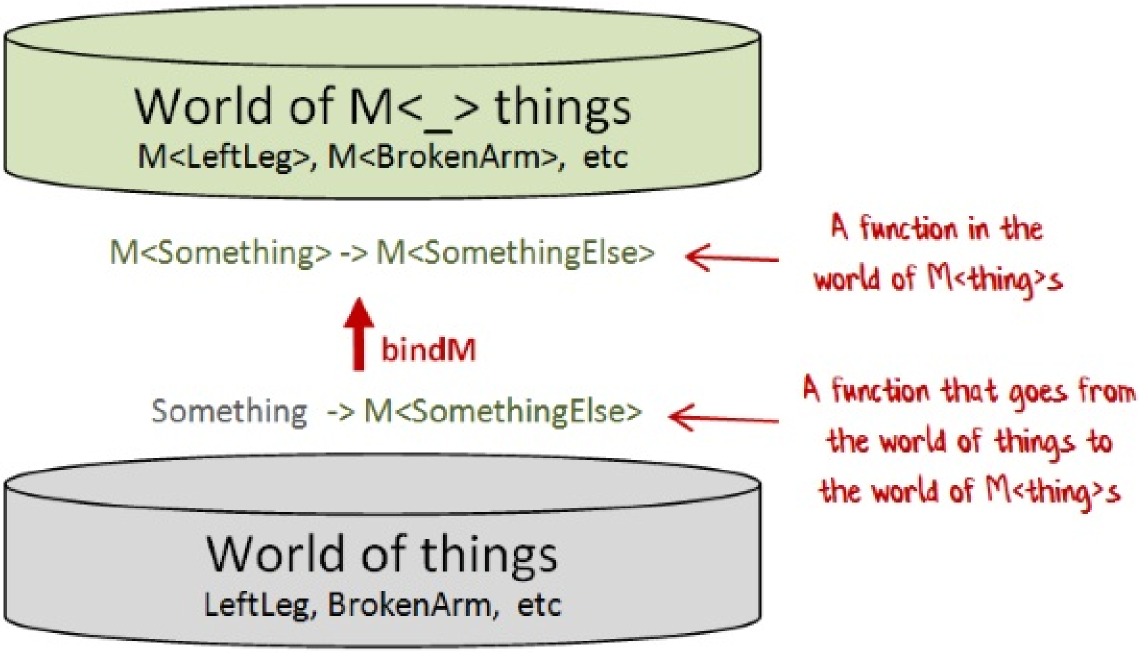
// create a beating heart generator (M<BeatingHeart>)
// from liveHeartM and the makeBeatingHeart function
let beatingHeartM = bindM makeBeatingHeart liveHeartM
```

There are a lot of intermediate values in there, and it can be made simpler by using piping, like this:

```
let beatingHeartM =
   DeadHeart "Anne"
   |> makeLiveHeart
   |> bindM makeBeatingHeart
```

The importance of bind

One way of thinking about `bindM` is that it is another "function converter", just like `mapM`. That is, given any "M-returning" function, it converts it to a function where the input and output are both `M` s.



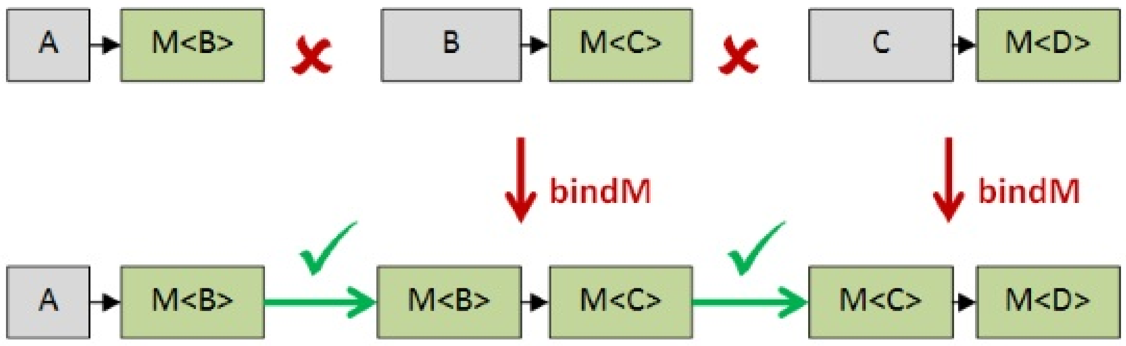
Just like `map`, `bind` appears in many other contexts.

For example, `Option.bind` transforms a option-generating function (`'a -> 'b option`) into a function whose inputs and outputs are options. Similarly, `List.bind` transforms a list-generating function (`'a -> 'b list`) into a function whose inputs and outputs are lists.

And I discuss yet another version of bind at length in my talk on functional error handling.

The reason that bind is so important is that "M-returning" functions crop up a lot, and they cannot be chained together easily because the output of one step does not match the input of the the next step.

By using `bindM`, we can convert each step into a function where the input and output are both `M` s, and then they *can* be chained together.



Testing the beating heart

As always, we construct the recipe ahead of time, in this case, to make a `BeatingHeart`.

```
let beatingHeartM =
    DeadHeart "Anne"
    |> makeLiveHeart
    |> bindM makeBeatingHeart
```

When the vital force is available, we can run `beatingHeartM` with the vital force...

```
let vf = {units = 10}

let beatingHeart, remainingFromHeart = runM beatingHeartM vf
```

...and we get this result:

```
val beatingHeart : BeatingHeart =
    BeatingHeart (LiveHeart ("Anne",{units = 1;}),{units = 1;})

val remainingFromHeart : VitalForce =
    {units = 8;}
```

Note that the remaining vital force is eight units, as we used up two units doing two steps.

The Whole Body

Finally, we have all the parts we need to assemble a complete body.

Here is Dr Frankenfunctor's definition of a live body:

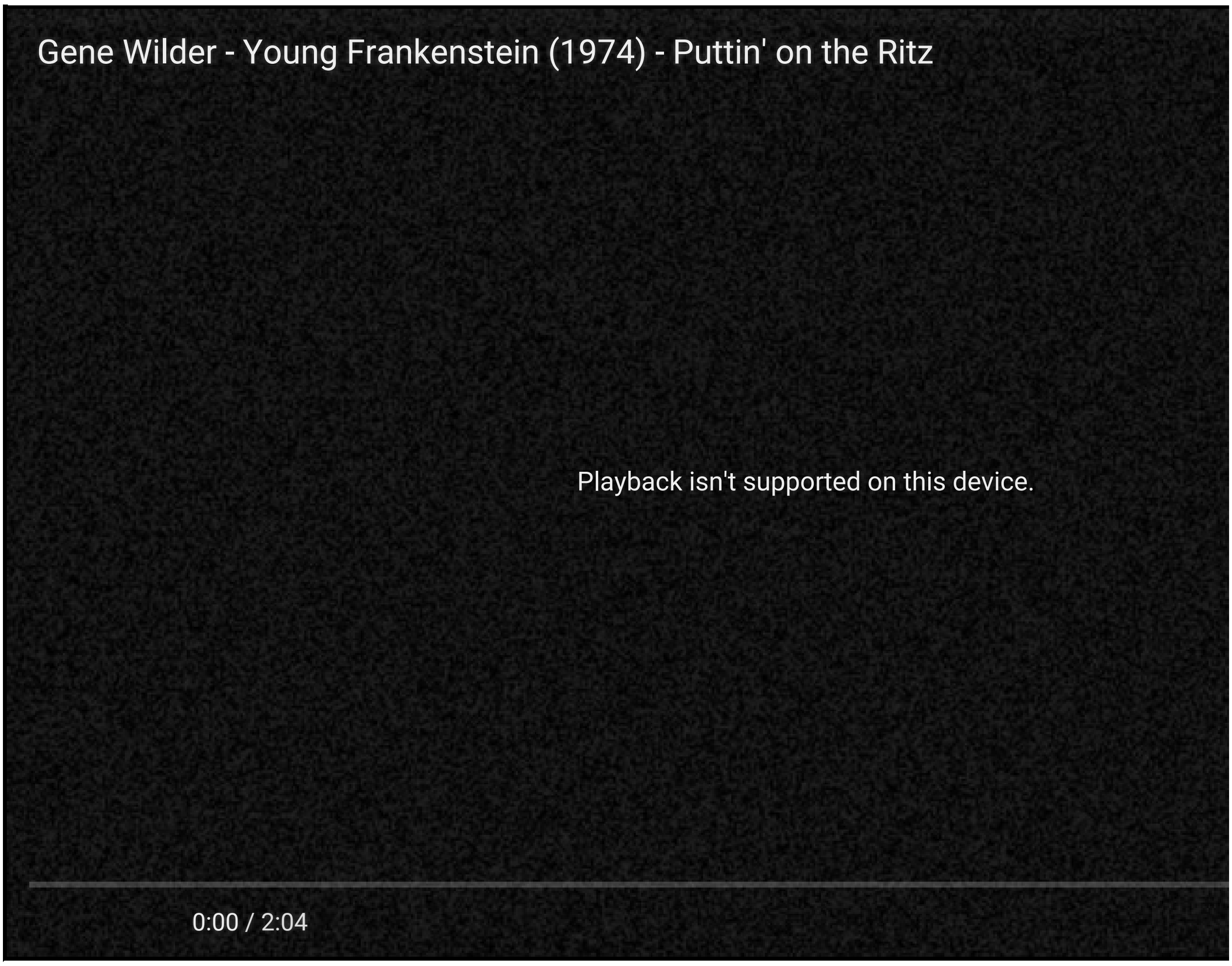
```
type LiveBody = {
    leftLeg: LiveLeftLeg
    rightLeg : LiveLeftLeg
    leftArm : LiveLeftArm
    rightArm : LiveRightArm
    head : LiveHead
    heart : BeatingHeart
    }
```

You can see that it uses all the subcomponents that we have already developed.

Two left feet

Because there was no right leg available, Dr Frankenfunctor decided to take a short cut and use *two* left legs in the body, hoping that no one would notice.

The result of this was that the creature had two left feet, which is not always a handicap, and indeed, the creature not only overcame this disadvantage but became a creditable dancer, as can be seen in this rare footage:



Assembling the subcomponents

The `LiveBody` type has six fields. How can we construct it from the various `M<BodyPart>` s that we have?

One way would be to repeat the technique that we used with `mapM` and `map2M`. We could create a `map3M` and `map4M` and so on.

For example, `map3M` could be defined like this:

```
let map3M f m1 m2 m3 =
    let becomeAlive vitalForce =
        let v1,remainingVitalForce = runM m1 vitalForce
        let v2,remainingVitalForce2 = runM m2 remainingVitalForce
        let v3,remainingVitalForce3 = runM m3 remainingVitalForce2
        let v4 = f v1 v2 v3
        v4, remainingVitalForce3
    M becomeAlive
```

But that gets tedious quite quickly. Is there a better way?

Why, yes there is!

To understand it, remember that record types like `LiveBody` have to be built all-or-nothing, but *functions* can be assembled step by step, thanks to the magic of currying and partial application.

So if we have a six parameter function that creates a `LiveBody`, like this:

```
val createBody :
    leftLeg:LiveLeftLeg ->
    rightLeg:LiveLeftLeg ->
    leftArm:LiveLeftArm ->
    rightArm:LiveRightArm ->
    head:LiveHead ->
    beatingHeart:BeatingHeart ->
    LiveBody
```

we can actually think of it as a *one* parameter function that returns a five parameter function, like this:

```
val createBody :
    leftLeg:LiveLeftLeg -> (five param function)
```

and then when we apply the function to the first parameter ("leftLeg") we get back that five parameter function:

```
(six param function) apply (first parameter) returns (five param function)
```

where the five parameter function has the signature:

```
    rightLeg:LiveLeftLeg ->
    leftArm:LiveLeftArm ->
    rightArm:LiveRightArm ->
    head:LiveHead ->
    beatingHeart:BeatingHeart ->
    LiveBody
```

This five parameter function can in turn be thought of as a one parameter function that returns a four parameter function:

```
    rightLeg:LiveLeftLeg -> (four parameter function)
```

Again, we can apply a first parameter ("rightLeg") and get back that four parameter function:

```
(five param function) apply (first parameter) returns (four param function)
```

where the four parameter function has the signature:

```
    leftArm:LiveLeftArm ->
    rightArm:LiveRightArm ->
    head:LiveHead ->
    beatingHeart:BeatingHeart ->
    LiveBody
```

And so on and so on, until eventually we get a function with one parameter. The function will have the signature `BeatingHeart -> LiveBody` .

When we apply the final parameter ("beatingHeart") then we get back our completed `LiveBody` .

We can use this trick for M-things as well!

We start with the six parameter function wrapped in an M, and an M parameter.

Let's assume there is some way to "apply" the M-function to the M-parameter. We should get back a five parameter function wrapped in an `M` .

```
// normal version
(six param function) apply (first parameter) returns (five param function)

// M-world version
M<six param function> applyM M<first parameter> returns M<five param function>
```

And then doing that again, we can apply the next M-parameter

```
// normal version
(five param function) apply (first parameter) returns (four param function)

// M-world version
M<five param function> applyM M<first parameter> returns M<four param function>
```

and so on, applying the parameters one by one until we get the final result.

Introducing applyM

This `applyM` function will have two parameters then, a function wrapped in an M, and a parameter wrapped in an M. The output will be the result of the function wrapped in an M.

Here's the implementation:

```
let applyM mf mx =
    let becomeAlive vitalForce =
        let f,remainingVitalForce = runM mf vitalForce
        let x,remainingVitalForce2 = runM mx remainingVitalForce
        let y = f x
        y, remainingVitalForce2
    M becomeAlive
```

As you can see it is quite similar to `map2M`, except that the "f" comes from unwrapping the first parameter itself.

Let's try it out!

First we need our six parameter function:

```
let createBody leftLeg rightLeg leftArm rightArm head beatingHeart =
    {
    leftLeg = leftLeg
    rightLeg = rightLeg
    leftArm = leftArm
    rightArm = rightArm
    head = head
    heart = beatingHeart
    }
```

And we're going to need to clone the left leg to use it for the right leg:

```
let rightLegM = leftLegM
```

Next, we need to wrap this `createBody` function in an `M`. How can we do that?

With the `returnM` function we defined earlier for skull, of course!

So putting it together, we have this code:

```
// move createBody to M-world -- a six parameter function wrapped in an M
let fSixParamM = returnM createBody

// apply first M-param to get a five parameter function wrapped in an M
let fFiveParamM = applyM fSixParamM leftLegM

// apply second M-param to get a four parameter function wrapped in an M
let fFourParamM = applyM fFiveParamM rightLegM

// etc
let fThreeParamM = applyM fFourParamM leftArmM
let fTwoParamM = applyM fThreeParamM rightArmM
let fOneParamM = applyM fTwoParamM headM

// after last application, the result is a M<LiveBody>
let bodyM = applyM fOneParamM beatingHeartM
```

It works! The result is a `M<LiveBody>` just as we want.

But that code sure is ugly! What can we do to make it look nicer?

One trick is to turn `applyM` into an infix operation, just like normal function application. The operator used for this is commonly written `<*>`.

```
let (<*>) = applyM
```

With this in place, we can rewite the above code as:

```
let bodyM =
    returnM createBody
    <*> leftLegM
    <*> rightLegM
    <*> leftArmM
    <*> rightArmM
    <*> headM
    <*> beatingHeartM
```

which is much nicer!

Another trick is to notice that the `returnM` followed by `applyM` is the same as `mapM`. So if we create an infix operator for `mapM` too...

```
let (<!>) = mapM
```

...we can get rid of the `returnM` as well and write the code like this:

```
let bodyM =
    createBody
    <!> leftLegM
    <*> rightLegM
    <*> leftArmM
    <*> rightArmM
    <*> headM
    <*> beatingHeartM
```

What's nice about this is that it reads almost as if you were just calling the original function (once you get used to the symbols!)

Testing the whole body

As always, we want to construct the recipe ahead of time. In this case, we have already created the `bodyM` that will give us a complete `LiveBody` when the vital force arrives.

Now all we have to do is wait for lightning to strike and charge the machinery that generates the vital force!



Source: Misfit Robot Daydream

Here it comes -- the vital force is available! Quickly we run `bodyM` in the usual way...

```
let vf = {units = 10}

let liveBody, remainingFromBody = runM bodyM vf
```

...and we get this result:

```
val liveBody : LiveBody =
  {leftLeg = LiveLeftLeg ("Boris",{units = 1;});
   rightLeg = LiveLeftLeg ("Boris",{units = 1;});
   leftArm = LiveLeftArm ("Victor",{units = 1;});
   rightArm = {lowerArm = LiveRightLowerArm ("Tom",{units = 1;});
               upperArm = LiveRightUpperArm ("Jerry",{units = 1;});};
   head = {brain = LiveBrain ("Abby Normal",{units = 1;});
           skull = Skull "Yorick";};
   heart = BeatingHeart (LiveHeart ("Anne",{units = 1;}),{units = 1;});}

val remainingFromBody : VitalForce = {units = 2;}
```

It's alive! We have successfully reproduced Dr Frankenfunctor's work!

Note that the body contains all the correct subcomponents, and that the remaining vital force has been correctly reduced to two units, as we used up eight units creating the body.

Summary

In this post, we extended our repertoire of manipulation techniques to include:

- `returnM` for the skull
- `bindM` for the beating heart
- `applyM` to assemble the whole body

The code samples used in this post are available on GitHub.

Next time

In the final installment, we'll refactor the code and review all the techniques used.

Refactoring the Monadster

UPDATE: Slides and video from my talk on this topic

Warning! This post contains gruesome topics, strained analogies, discussion of monads

Welcome to the third installment in the gripping tale of Dr Frankenfunctor and the Monadster!

We saw in the first installment how Dr Frankenfunctor created life out of dead body parts using "Monadster part generators" (or "M"s for short), that would, on being supplied with some vital force, return a live body part.

We also saw how the leg and arms of the creature were created, and how these M-values could be processed and combined using `mapM` and `map2M`.

In the second installment we learned how the head, heart and body were built using other powerful techniques such as `returnM`, `bindM` and `applyM`.

In this last installment, we'll review all the techniques used, refactor the code, and compare Dr Frankenfunctor's techniques to the modern-day state monad.

Links for the complete series:

- Part 1 - Dr Frankenfunctor and the Monadster
- Part 2 - Completing the body
- Part 3 - Review and refactoring (*this post*)

Review of the the techniques used

Before we refactor, let's review all the techniques we have used.

The M type

We couldn't create an actual live body part until the vital force was available, yet we wanted to manipulate them, combine them, etc. *before* the lightning struck. We did this by creating a type `M` that wrapped a "become alive" function for each part. We could then think of `M<BodyPart>` as a *recipe* or *instructions* for creating a `BodyPart` when the time came.

The definition of `M` was:

```
type M<'a> = M of (VitalForce -> 'a * VitalForce)
```

mapM

Next, we wanted to transform the contents of an `M` without using any vital force. In our specific case, we wanted to turn a broken arm recipe (`M<BrokenLeftArm>`) into a unbroken arm recipe (`M<LeftArm>`). The solution was to implement a function `mapM` that took a normal function `'a -> 'b` and turned it into a `M<'a> -> M<'b>` function.

The signature of `mapM` was:

```
val mapM : f:('a -> 'b) -> M<'a> -> M<'b>
```

map2M

We also wanted to combine two M-recipes to make a new one. In that particular case, it was combining an upper arm (`M<UpperRightArm>`) and lower arm (`M<LowerRightArm>`) into a whole arm (`M<RightArm>`). The solution was `map2M` .

The signature of `map2M` was:

```
val map2M : f:('a -> 'b -> 'c) -> M<'a> -> M<'b> -> M<'c>
```

returnM

Another challenge was to lift a normal value into the world of M-recipes directly, without using any vital force. In that particular case, it was turning a `Skull` into an `M<Skull>` so it could be used with `map2M` to make a whole head.

The signature of `returnM` was:

```
val returnM : 'a -> M<'a>
```

Monadic functions

We created many functions that had a similar shape. They all take something as input and return a M-recipe as output. In other words, they have this signature:

```
val monadicFunction : 'a -> M<'b>
```

Here are some examples of actual monadic functions that we used:

```
val makeLiveLeftLeg : DeadLeftLeg -> M<LiveLeftLeg>
val makeLiveRightLowerArm : DeadRightLowerArm -> M<LiveRightLowerArm>
val makeLiveHeart : DeadHeart -> M<LiveHeart>
val makeBeatingHeart : LiveHeart -> M<BeatingHeart>
// and also
val returnM : 'a -> M<'a>
```

bindM

The functions up to now did not require access to the vital force. But then we found that we needed to chain two monadic functions together. In particular, we needed to chain the output of `makeLiveHeart` (with signature `DeadHeart -> M<LiveHeart>`) to the input of `makeBeatingHeart` (with signature `LiveHeart -> M<BeatingHeart>`). The solution was `bindM`, which transforms monadic functions of the form `'a -> M<'b>` into functions in the M-world (`M<'a> -> M<'b>`) which could then be composed together.

The signature of `bindM` was:

```
val bindM : f:('a -> M<'b>) -> M<'a> -> M<'b>
```

applyM

Finally, we needed a way to combine a large number of M-parameters to make the live body. Rather than having to create special versions of map (`map4M`, `map5M`, `map6M`, etc), we implemented a generic `applyM` function that could apply an M-function to an M-parameter. From that, we could work with a function of any size, step by step, using partial application to apply one M-parameter at a time.

The signature of `applyM` was:

```
val applyM : M<('a -> 'b)> -> M<'a> -> M<'b>
```

Defining the others functions in terms of bind and return

Note that of all these functions, only `bindM` needed access to the vital force.

In fact, as we'll see below, the functions `mapM`, `map2M`, and `applyM` can actually be defined in terms of `bindM` and `returnM`!

Refactoring to a computation expression

A lot of the functions we have created have a very similar shape, resulting in a lot of duplication. Here's one example:

```
let makeLiveLeftLegM deadLeftLeg  =
    let becomeAlive vitalForce =
        let (DeadLeftLeg label) = deadLeftLeg
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveLeftLeg = LiveLeftLeg (label,oneUnit)
        liveLeftLeg, remainingVitalForce
    M becomeAlive  // wrap the function in a single case union
```

In particular, there is a lot of explicit handling of the vital force.

In most functional languages, there is a way to hide this so that the code looks much cleaner.

In Haskell, developers use "do-notation", in Scala people use "for-yield" (the "for comprehension"). And in F#, people use computation expressions.

To create a computation expression in F#, you just need two things to start with, a "bind" and a "return", both of which we have.

Next, you define a class with specially named methods `Bind` and `Return` :

```
type MonsterBuilder()=
    member this.Return(x) = returnM x
    member this.Bind(xM,f) = bindM f xM
```

And finally, you create an instance of this class:

```
let monster = new MonsterBuilder()
```

When this is done, we have access to the special syntax `monster {...}` , just like `async{...}` , `seq{...}` , etc.

- The `let! x = xM` syntax requires that the right side is an M-type, say `M<X>` . `let!` unwraps the `M<X>` into an `x` and binds it to the left side -- "x" in this case.
- The `return y` syntax requires that return value is a "normal" type, `Y` say. `return` wraps it into a `M<Y>` (using `returnM`) and returns it as the overall value of the `monster` expression.

So some example code would look like this:

```
monster {
    let! x = xM  // unwrap an M<X> into an X and bind to "x"
    return y     // wrap a Y and return an M<Y>
    }
```

If you want more on computation expressions, I have an in-depth series of posts about them.

Redefining mapM and friends

With `monster` expressions available, let's rewrite `mapM` and the other functions.

mapM

`mapM` takes a function and a wrapped M-value and returns the function applied to the inner value.

Here is an implementation using `monster`:

```
let mapM f xM =
    monster {
        let! x = xM  // unwrap the M<X>
        return f x   // return M of (f x)
        }
```

If we compile this implementation, we get the same signature as the previous implementation:

```
val mapM : f:('a -> 'b) -> M<'a> -> M<'b>
```

map2M

`map2M` takes a function and two wrapped M-values and returns the function applied to both the values.

It's also easy to write using `monster` expressions:

```
let map2M f xM yM =
    monster {
        let! x = xM  // unwrap M<X>
        let! y = yM  // unwrap M<Y>
        return f x y // return M of (f x y)
        }
```

If we compile this implementation, we again get the same signature as the previous implementation:

```
val map2M : f:('a -> 'b -> 'c) -> M<'a> -> M<'b> -> M<'c>
```

applyM

`applyM` takes a wrapped function and a wrapped value and returns the function applied to the value.

Again, it's trivial to write using `monster` expressions:

```
let applyM fM xM =
    monster {
        let! f = fM  // unwrap M<F>
        let! x = xM  // unwrap M<X>
        return f x   // return M of (f x)
        }
```

And the signature is as expected

```
val applyM : M<('a -> 'b)> -> M<'a> -> M<'b>
```

Manipulating the vital force from within a monster context

We'd like to use the monster expression to rewrite all our other functions too, but there is a stumbling block.

Many of our functions have a body that looks like this:

```
// extract a unit of vital force from the context
let oneUnit, remainingVitalForce = getVitalForce vitalForce

// do something

// return value and remaining vital force
liveBodyPart, remainingVitalForce
```

In other words, we're *getting* some of the vital force and then *putting* a new vital force to be used for the next step.

We are familiar with "getters" and "setters" in object-oriented programming, so let's see if we can write similar ones that will work in the `monster` context.

Introducing getM

Let's start with the getter. How should we implement it?

Well, the vital force is only available in the context of becoming alive, so the function must follow the familiar template:

```
let getM =
    let doSomethingWhileLive vitalForce =
        // what here ??
        what to return??, vitalForce
    M doSomethingWhileLive
```

Note that getting the `vitalForce` doesn't use any up, so the original amount can be returned untouched.

But what should happen in the middle? And what should be returned as the first element of the tuple?

The answer is simple: just return the vital force itself!

```
let getM =
    let doSomethingWhileLive vitalForce =
        // return the current vital force in the first element of the tuple
        vitalForce, vitalForce
    M doSomethingWhileLive
```

`getM` is a `M<VitalForce>` value, which means that we can unwrap it inside a monster expression like this:

```
monster {
    let! vitalForce = getM
    // do something with vital force
    }
```

Introducing putM

For the putter, the implementation is a function with a parameter for the new vital force.

```
let putM newVitalForce  =
    let doSomethingWhileLive vitalForce =
        what here ??
    M doSomethingWhileLive
```

Again, what should we do in the middle?

The most important thing is that the `newVitalForce` becomes the value that is passed on to the next step. We must throw away the original vital force!

Which in turn means that `newVitalForce` *must* be used as the second part of the tuple that is returned.

And what should be in the first part of the tuple that is returned? There is no sensible value to return, so we'll just use `unit`.

Here's the final implementation:

```
let putM newVitalForce  =
    let doSomethingWhileLive vitalForce =
        // return nothing in the first element of the tuple
        // return the newVitalForce in the second element of the tuple
        (), newVitalForce
    M doSomethingWhileLive
```

With `getM` and `putM` in place, we can now create a function that

- gets the current vital force from the context
- extracts one unit from that
- replaces the current vital force with the remaining vital force
- returns the one unit of vital force to the caller

And here's the code:

```
let useUpOneUnitM =
    monster {
        let! vitalForce = getM
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        do! putM remainingVitalForce
        return oneUnit
        }
```

Using the monster expression to rewrite all the other functions

With `useUpOneUnitM`, we can start to rewrite all the other functions.

For example, the original function `makeLiveLeftLegM` looked like this, with lots of explicit handling of the vital force.

```
let makeLiveLeftLegM deadLeftLeg  =
    let becomeAlive vitalForce =
        let (DeadLeftLeg label) = deadLeftLeg
        let oneUnit, remainingVitalForce = getVitalForce vitalForce
        let liveLeftLeg = LiveLeftLeg (label,oneUnit)
        liveLeftLeg, remainingVitalForce
    M becomeAlive  // wrap the function in a single case union
```

The new version, using a monster expression, has implicit handling of the vital force, and consequently looks much cleaner.

```
let makeLiveLeftLegM deadLeftLeg =
    monster {
        let (DeadLeftLeg label) = deadLeftLeg
        let! oneUnit = useUpOneUnitM
        return LiveLeftLeg (label,oneUnit)
        }
```

Similarly, we can rewrite all the arm surgery code like this:

```
let makeLiveRightLowerArm (DeadRightLowerArm label) =
    monster {
        let! oneUnit = useUpOneUnitM
        return LiveRightLowerArm (label,oneUnit)
        }

let makeLiveRightUpperArm (DeadRightUpperArm label) =
    monster {
        let! oneUnit = useUpOneUnitM
        return LiveRightUpperArm (label,oneUnit)
        }

// create the M-parts
let lowerRightArmM = DeadRightLowerArm "Tom" |> makeLiveRightLowerArm
let upperRightArmM = DeadRightUpperArm "Jerry" |> makeLiveRightUpperArm

// turn armSurgery into an M-function
let armSurgeryM  = map2M armSurgery

// do surgery to combine the two M-parts into a new M-part
let rightArmM = armSurgeryM lowerRightArmM upperRightArmM
```

And so on. This new code is much cleaner.

In fact, we can make it cleaner yet by eliminating the intermediate values such as `armSurgery` and `armSurgeryM` and putting everything in the one monster expression.

```
let rightArmM = monster {
    let! lowerArm = DeadRightLowerArm "Tom" |> makeLiveRightLowerArm
    let! upperArm = DeadRightUpperArm "Jerry" |> makeLiveRightUpperArm
    return {lowerArm=lowerArm; upperArm=upperArm}
    }
```

We can use this approach for the head as well. We don't need `headSurgery` or `returnM` any more.

```
let headM = monster {
    let! brain = makeLiveBrain deadBrain
    return {brain=brain; skull=skull}
    }
```

Finally, we can use a `monster` expression to create the whole body too:

```
// a function to create a M-body given all the M-parts
let createBodyM leftLegM rightLegM leftArmM rightArmM headM beatingHeartM =
    monster {
        let! leftLeg = leftLegM
        let! rightLeg = rightLegM
        let! leftArm = leftArmM
        let! rightArm = rightArmM
        let! head = headM
        let! beatingHeart = beatingHeartM

        // create the record
        return {
            leftLeg = leftLeg
            rightLeg = rightLeg
            leftArm = leftArm
            rightArm = rightArm
            head = head
            heart = beatingHeart
            }
        }

// create the M-body
let bodyM = createBodyM leftLegM rightLegM leftArmM rightArmM headM beatingHeartM
```

NOTE: The complete code using `monster` expressions is available on GitHub.

Monster expressions vs applyM

We previously used an alternative way to create the body, using `applyM`.

For reference, here's that way using `applyM`:

```
let createBody leftLeg rightLeg leftArm rightArm head beatingHeart =
    {
    leftLeg = leftLeg
    rightLeg = rightLeg
    leftArm = leftArm
    rightArm = rightArm
    head = head
    heart = beatingHeart
    }

let bodyM =
    createBody
    <!> leftLegM
    <*> rightLegM
    <*> leftArmM
    <*> rightArmM
    <*> headM
    <*> beatingHeartM
```

So what's the difference?

Aesthetically there is a bit of difference, but you could legitimately prefer either.

However, there is a much more important difference between the `applyM` approach and the `monster` expression approach.

The `applyM` approach allows the parameters to be run *independently* or *in parallel*, while the `monster` expression approach requires that the parameters are run *in sequence*, with the output of one being fed into the input of the next.

That's not relevant for this scenario, but can be important for other situations such as validation or async. For example, in a validation context, you may want to collect all the validation errors at once, rather than only returning the first one that fails.

Relationship to the state monad

Dr Frankenfunctor was a pioneer in her time, blazing a new trail, but she did not generalize her discoveries to other domains.

Nowadays, this pattern of threading some information through a series of functions is very common, and we give it a standard name: "State Monad".

Now to be a true monad, there are various properties that must be satisfied (the so-called monad laws), but I am not going to discuss them here, as this post is not intended to be a monad tutorial.

Instead, I'll just focus on how the state monad might be defined and used in practice.

First off, to be truly reusable, we need to replace the `VitalForce` type with other types. So our function-wrapping type (call it `S`) must have *two* type parameters, one for the type of the state, and one for the type of the value.

```
type S<'State,'Value> =
    S of ('State -> 'Value * 'State)
```

With this defined, we can create the usual suspects: `runS`, `returnS` and `bindS`.

```
// encapsulate the function call that "runs" the state
let runS (S f) state = f state

// lift a value to the S-world
let returnS x =
    let run state =
        x, state
    S run

// lift a monadic function to the S-world
let bindS f xS =
    let run state =
        let x, newState = runS xS state
        runS (f x) newState
    S run
```

Personlly, I'm glad that we got to understood how these worked in the `M` context before making them completely generic. I don't know about you, but signatures like these

```
val runS : S<'a,'b> -> 'a -> 'b * 'a
val bindS : f:('a -> S<'b,'c>) -> S<'b,'a> -> S<'b,'c>
```

would be really hard to understand on their own, without any preparation.

Anyway, with those basics in place we can create a `state` expression.

```
type StateBuilder()=
    member this.Return(x) = returnS x
    member this.Bind(xS,f) = bindS f xS

let state = new StateBuilder()
```

`getS` and `putS` are defined in a similar way as `getM` and `putM` for `monster`.

```
let getS =
    let run state =
        // return the current state in the first element of the tuple
        state, state
    S run
// val getS : S<State>

let putS newState =
    let run _ =
        // return nothing in the first element of the tuple
        // return the newState in the second element of the tuple
        (), newState
    S run
// val putS : 'State -> S<unit>
```

Property based testing of the state expression

Before moving on, how do we know that our `state` implementation is correct? What does it even *mean* to be correct?

Well, rather than writing lots of example based tests, this is a great candidate for a property-based testing approach.

The properties we might expect to be satisfied include:

- **The monad laws**.
- **Only the last put counts**. That is, putting X then putting Y should be the same as just putting Y.
- **Get should return the last put**. That is, putting X then doing a get should return the same X.

and so on.

I won't go into this any more right now. I suggest watching the talk for a more in-depth discussion.

Using the state expression instead of the monster expression

We can now use `state` expressions exactly as we did `monster` expressions. Here's an example:

```
// combine get and put to extract one unit
let useUpOneUnitS = state {
    let! vitalForce = getS
    let oneUnit, remainingVitalForce = getVitalForce vitalForce
    do! putS remainingVitalForce
    return oneUnit
    }

type DeadLeftLeg = DeadLeftLeg of Label
type LiveLeftLeg = LiveLeftLeg of Label * VitalForce

// new version with implicit handling of vital force
let makeLiveLeftLeg (DeadLeftLeg label) = state {
    let! oneUnit = useUpOneUnitS
    return LiveLeftLeg (label,oneUnit)
    }
```

Another example is how to build a `BeatingHeart` :

```
type DeadHeart = DeadHeart of Label
type LiveHeart = LiveHeart of Label * VitalForce
type BeatingHeart = BeatingHeart of LiveHeart * VitalForce

let makeLiveHeart (DeadHeart label) = state {
    let! oneUnit = useUpOneUnitS
    return LiveHeart (label,oneUnit)
    }

let makeBeatingHeart liveHeart = state {
    let! oneUnit = useUpOneUnitS
    return BeatingHeart (liveHeart,oneUnit)
    }

let beatingHeartS = state {
    let! liveHeart = DeadHeart "Anne" |> makeLiveHeart
    return! makeBeatingHeart liveHeart
    }

let beatingHeart, remainingFromHeart = runS beatingHeartS vf
```

As you can see, the `state` expression automatically picked up that `VitalForce` was being used as the state -- we didn't need to specify it explicitly.

So, if you have a `state` expression type available to you, you don't need to create your own expressions like `monster` at all!

For a more detailed and complex example of the state monad in F#, check out the FSharpx library.

NOTE: The complete code using `state` *expressions is available on GitHub.*

Other examples of using a state expression

The state computation expression, once defined, can be used for all sorts of things. For example, we can use `state` to model a stack.

Let's start by defining a `Stack` type and associated functions:

```
// define the type to use as the state
type Stack<'a> = Stack of 'a list

// define pop outside of state expressions
let popStack (Stack contents) =
    match contents with
    | [] -> failwith "Stack underflow"
    | head::tail ->
        head, (Stack tail)

// define push outside of state expressions
let pushStack newTop (Stack contents) =
    Stack (newTop::contents)

// define an empty stack
let emptyStack = Stack []

// get the value of the stack when run
// starting with the empty stack
let getValue stackM =
    runS stackM emptyStack |> fst
```

Note that none of that code knows about or uses the `state` computation expression.

To make it work with `state`, we need to define a customized getter and putter for use in a `state` context:

```
let pop() = state {
    let! stack = getS
    let top, remainingStack = popStack stack
    do! putS remainingStack
    return top
    }

let push newTop = state {
    let! stack = getS
    let newStack = pushStack newTop stack
    do! putS newStack
    return ()
    }
```

With these in place we can start coding our domain!

Stack-based Hello World

Here's a simple one. We push "world", then "hello", then pop the stack and combine the results.

```
let helloWorldS = state {
    do! push "world"
    do! push "hello"
    let! top1 = pop()
    let! top2 = pop()
    let combined = top1 + " " + top2
    return combined
    }

let helloWorld = getValue helloWorldS // "hello world"
```

Stack-based calculator

Here's a simple stack-based calculator:

```
let one = state {do! push 1}
let two = state {do! push 2}

let add = state {
    let! top1 = pop()
    let! top2 = pop()
    do! push (top1 + top2)
    }
```

And now we can combine these basic states to build more complex ones:

```
let three = state {
    do! one
    do! two
    do! add
    }

let five = state {
    do! two
    do! three
    do! add
    }
```

Remember that, just as with the vital force, all we have now is a *recipe* for building a stack. We still need to *run* it to execute the recipe and get the result.

Let's add a helper to run all the operations and return the top of the stack:

```
let calculate stackOperations = state {
    do! stackOperations
    let! top = pop()
    return top
    }
```

Now we can evaluate the operations, like this:

```
let threeN = calculate three |> getValue // 3

let fiveN = calculate five |> getValue   // 5
```

OK, OK, some monad stuff

People always want to know about monads, even though I do not want these posts to degenerate into yet another monad tutorial.

So here's how they fit in with what we have worked with in these posts.

A **functor** (in a programming sense, anyway) is a data structure (such as Option, or List, or State) which has a `map` function associated with it. And the `map` function has some properties that it must satisfy (the "functor laws").

A **applicative functor** (in a programming sense) is a data structure (such as Option, or List, or State) which has two functions associated with it: `apply` and `pure` (which is the same as `return`). And these functions have some properties that they must satisfy (the "applicative functor laws").

Finally, a **monad** (in a programming sense) is a data structure (such as Option, or List, or State) which has two functions associated with it: `bind` (often written as `>>=`) and `return`. And again, these functions have some properties that they must satisfy (the "monad laws").

Of these three, the monad is most "powerful" in a sense, because the `bind` function allows you to chain M-producing functions together, and as we have seen, `map` and `apply` can be written in terms of `bind` and `return`.

So you can see that both our original `M` type and the more generic `State` type, in conjunction with their supporting functions, are monads, (assuming that our `bind` and `return` implementations satisfy the monad laws).

For a visual version of these definitions, there is a great post called Functors, Applicatives, And Monads In Pictures.

Further reading

There are a lot of posts about state monads on the web, of course. Most of them are Haskell oriented, but I hope that those explanations will make more sense after reading this series of posts, so I'm only going to mention a few follow up links.

- State monad in pictures
- "A few monads more", from "Learn You A Haskell"
- Much Ado About Monads. Discussion about state monad in F#.

And for another important use of "bind", you might find my talk on functional error handling useful.

If you want to see F# implementations of other monads, look no further than the FSharpx project.

Summary

Dr Frankenfunctor was a groundbreaking experimentalist, and I'm glad that I have been able to share insights on her way of working.

We've seen how she discovered a primitive monad-like type, `M<BodyPart>`, and how `mapM`, `map2M`, `returnM`, `bindM` and `applyM` were all developed to solve specific problems.

We've also seen how the need to solve the same problems led to the modern-day state monad and computation expression.

Anyway, I hope that this series of posts has been enlightening. My not-so-secret wish is that monads and their associated combinators will no longer be so shocking to you now...



...and that you can use them wisely in your own projects. Good luck!

NOTE: The code samples used in this post are available on GitHub.

In this series of posts, I'll attempt to describe some of the core functions for dealing with generic data types (such as `Option` and `List`). This is a follow-up post to my talk on functional patterns.

Yes, I know that I promised not to do this kind of thing, but for this post I thought I'd take a different approach from most people. Rather than talking about abstractions such as type classes, I thought it might be useful to focus on the core functions themselves and how they are used in practice.

In other words, a sort of "man page" for `map`, `return`, `apply`, and `bind`.

So, there is a section for each function, describing their name (and common aliases), common operators, their type signature, and then a detailed description of why they are needed and how they are used, along with some visuals (which I always find helpful).

- Understanding map and apply. A toolset for working with elevated worlds.
- Understanding bind. Or, how to compose world-crossing functions.
- Using the core functions in practice. Working with independent and dependent data.
- Understanding traverse and sequence. Mixing lists and elevated values.
- Using map, apply, bind and sequence in practice. A real-world example that uses all the techniques.
- Reinventing the Reader monad. Or, designing your own elevated world.
- Map and Bind and Apply, a summary. .

Understanding map and apply

In this series of posts, I'll attempt to describe some of the core functions for dealing with generic data types (such as `Option` and `List`). This is a follow-up post to my talk on functional patterns.

Yes, I know that I promised not to do this kind of thing, but for this post I thought I'd take a different approach from most people. Rather than talking about abstractions such as type classes, I thought it might be useful to focus on the core functions themselves and how they are used in practice.

In other words, a sort of "man page" for `map`, `return`, `apply`, and `bind`.

So, there is a section for each function, describing their name (and common aliases), common operators, their type signature, and then a detailed description of why they are needed and how they are used, along with some visuals (which I always find helpful).

Haskellers and category-theorists may want to look away now! There will be no mathematics and quite a lot of hand-waving. I am going to avoid jargon and Haskell-specific concepts such as type classes and focus on the big picture as much as possible. The concepts here should be applicable to any kind of functional programming in any language.

I know that some people might dislike this approach. That's fine. There is no shortage of more academic explanations on the web. Start with this and this.

Finally, as with most of the posts on this site, I am writing this up for my own benefit as well, as part of my own learning process. I don't claim to be an expert at all, so if I have made any errors please let me know.

Background

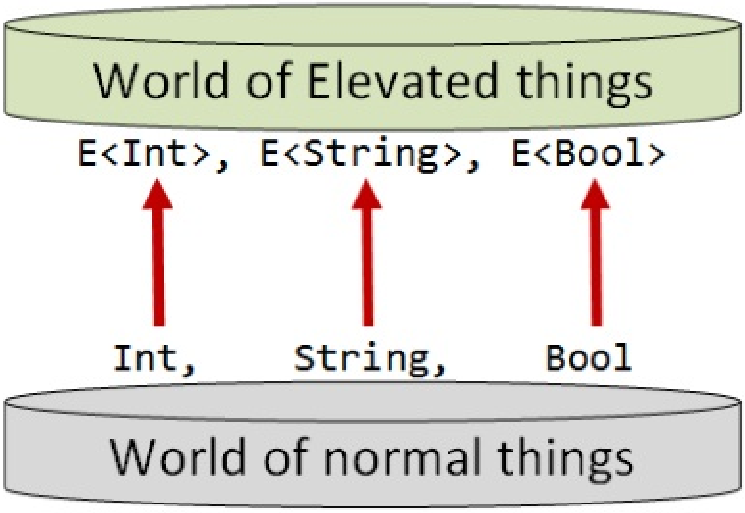
To start with, let me provide the background and some terminology.

Imagine that there are two worlds that we could program in: a "normal" everyday world and a world that I will call the "elevated world" (for reasons that I will explain shortly).

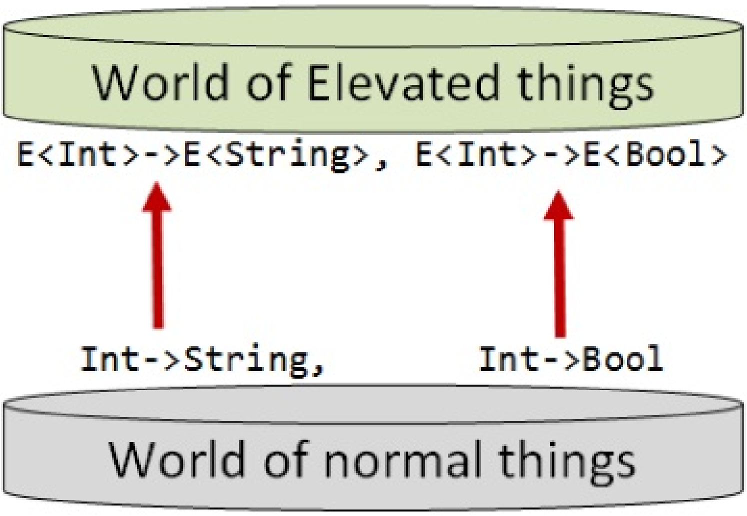
The elevated world is very similar to the normal world. In fact, every thing in the normal world has a corresponding thing in the elevated world.

So, for example, we have the set of values called `Int` in the normal world, and in the elevated world there is a parallel set of values called, say, `E<Int>`. Similarly, we have the set of values called `String` in the normal world, and in the elevated world there is a parallel

set of values called `E<String>`.



Also, just as there are functions between `Int`s and `String`s in the normal world, so there are functions between `E<Int>`s and `E<String>`s in the elevated world.



Note that I am deliberately using the term "world" rather than "type" to emphasis that the *relationships* between values in the world are just as important as the underlying data type.

What exactly is an elevated world?

I can't define what an elevated world is exactly, because there are too many different kinds of elevated worlds, and they don't have anything in common.

Some of them represent data structures (`Option<T>`), some of them represent workflows (`State<T>`), some of them represent signals (`Observable<T>`), or asychronous values (`Async<T>`), or other concepts.

But even though the various elevated worlds have nothing in common specifically, there *are* commonalities in the way they can be worked with. We find that certain issues occur over and over again in different elevated worlds, and we can use standard tools and patterns to deal with these issues.

The rest of this series will attempt to document these tools and patterns.

Series contents

This series is developed as follows:

- First, I'll discuss the tools we have for lifting normal things into the elevated world. This includes functions such as `map`, `return`, `apply` and `bind`.
- Next, I'll discuss how you can combine elevated values in different ways, based on whether the values are independent or dependent.
- Next, we'll look at some ways of mixing lists with other elevated values.
- Finally, we'll look at two real-world examples that put all these techniques to use, and we'll find ourselves accidentally inventing the Reader monad.

Here's a list of shortcuts to the various functions:

- **Part 1: Lifting to the elevated world**
  - The `map` function
  - The `return` function
  - The `apply` function
  - The `liftN` family of functions
  - The `zip` function and ZipList world
- **Part 2: How to compose world-crossing functions**
  - The `bind` function
  - List is not a monad. Option is not a monad.
- **Part 3: Using the core functions in practice**
  - Independent and dependent data
  - Example: Validation using applicative style and monadic style
  - Lifting to a consistent world
  - Kleisli world
- **Part 4: Mixing lists and elevated values**
  - Mixing lists and elevated values
  - The `traverse` / `MapM` function
  - The `sequence` function
  - "Sequence" as a recipe for ad-hoc implementations
  - Readability vs. performance

  - Dude, where's my `filter` ?
- **Part 5: A real-world example that uses all the techniques**
  - Example: Downloading and processing a list of websites
  - Treating two worlds as one
- **Part 6: Designing your own elevated world**
  - Designing your own elevated world
  - Filtering out failures
  - The Reader monad
- **Part 7: Summary**
  - List of operators mentioned
  - Further reading

Part 1: Lifting to the elevated world

The first challenge is: how do we get from the normal world to the elevated world?

First, we will assume that for any particular elevated world:

- Every type in the normal world has a corresponding type in the elevated world.
- Every value in the normal world has a corresponding value in the elevated world.
- Every function in the normal world has a corresponding function in the elevated world.

The concept of moving something from the normal world to the elevated world is called "lifting" (which is why I used the term "elevated world" in the first place).

We'll call these corresponding things "lifted types" and "lifted values" and "lifted functions".

Now because each elevated world is different, there is no common implementation for lifting, but we can give names to the various "lifting" patterns, such as `map` and `return` .

NOTE: There is no standard name for these lifted types. I have seen them called "wrapper types" or "augmented types" or "monadic types". I'm not really happy with any of these names, so I invented a new one! Also, I'm trying avoid any assumptions, so I don't want to imply that the lifted types are somehow better or contain extra information. I hope that by using the word "elevated" in this post, I can focus on the lifting process *rather than on the types themselves.*

As for using the word "monadic", that would be inaccurate, as there is no requirement that these types are part of a monad.

The `map` function

Common Names: `map`, `fmap`, `lift`, `Select`

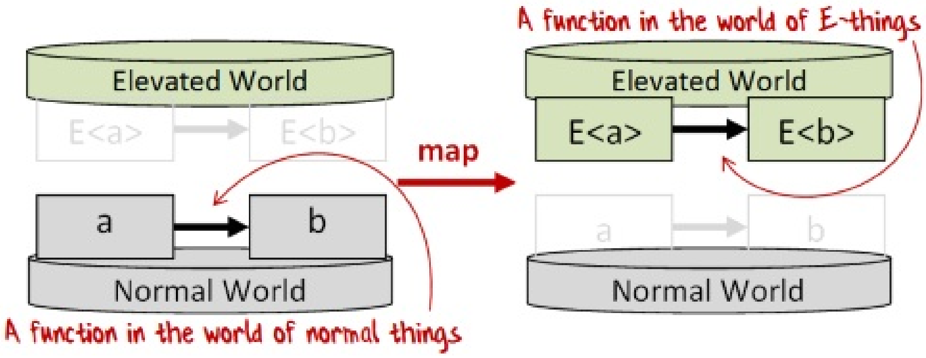
Common Operators: `<$>` `<!>`

What it does: Lifts a function into the elevated world

Signature: `(a->b) -> E<a> -> E<b>`. Alternatively with the parameters reversed: `E<a> -> (a->b) -> E<b>`

Description

"map" is the generic name for something that takes a function in the normal world and transforms it into a corresponding function in the elevated world.



Each elevated world will have its own implementation of map.

Alternative interpretation

An alternative interpretation of `map` is that it is a *two* parameter function that takes an elevated value (`E<a>`) and a normal function (`a->b`), and returns a new elevated value (`E<b>`) generated by applying the function `a->b` to the internal elements of `E<a>`.



In languages where functions are curried by default, such as F#, both these interpretation are the same. In other languages, you may need to curry/uncurry to switch between the two uses.

Note that the *two* parameter version often has the signature `E<a> -> (a->b) -> E<b>`, with the elevated value first and the normal function second. From an abstract point of view, there's no difference between them -- the map concept is the same -- but obviously, the parameter order affects how you might use map functions in practice.

Implementation examples

Here are two examples of how map can be defined for options and lists in F#.

```
/// map for Options
let mapOption f opt =
    match opt with
    | None ->
        None
    | Some x ->
        Some (f x)
// has type : ('a -> 'b) -> 'a option -> 'b option

/// map for Lists
let rec mapList f list =
    match list with
    | [] ->
        []
    | head::tail ->
        // new head + new tail
        (f head) :: (mapList f tail)
// has type : ('a -> 'b) -> 'a list -> 'b list
```

These functions are built-in of course, so we don't need to define them, I've done it just to show what they might look for some common types.

Usage examples

Here are some examples of how map can be used in F#:

```
// Define a function in the normal world
let add1 x = x + 1
// has type : int -> int

// A function lifted to the world of Options
let add1IfSomething = Option.map add1
// has type : int option -> int option

// A function lifted to the world of Lists
let add1ToEachElement = List.map add1
// has type : int list -> int list
```

With these mapped versions in place you can write code like this:

```
Some 2 |> add1IfSomething    // Some 3
[1;2;3] |> add1ToEachElement // [2; 3; 4]
```

In many cases, we don't bother to create an intermediate function -- partial application is used instead:

```
Some 2 |> Option.map add1    // Some 3
[1;2;3] |> List.map add1     // [2; 3; 4]
```

The properties of a correct map implementation

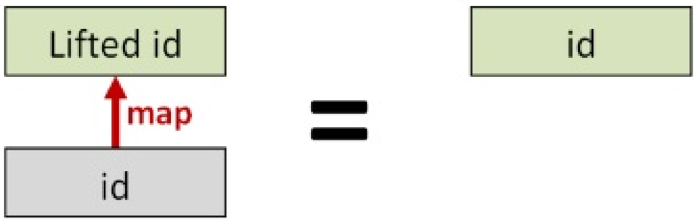
I said earlier that the elevated world is in some ways a mirror of the normal world. Every function in the normal world has a corresponding function in the elevated world, and so on. We want `map` to return this corresponding elevated function in a sensible way.

For example, `map` of `add` should not (wrongly) return the elevated version of `multiply`, and `map` of `lowercase` should not return the elevated version of `uppercase`! But how can we be *sure* that a particular implementation of map does indeed return the *correct* corresponding function?

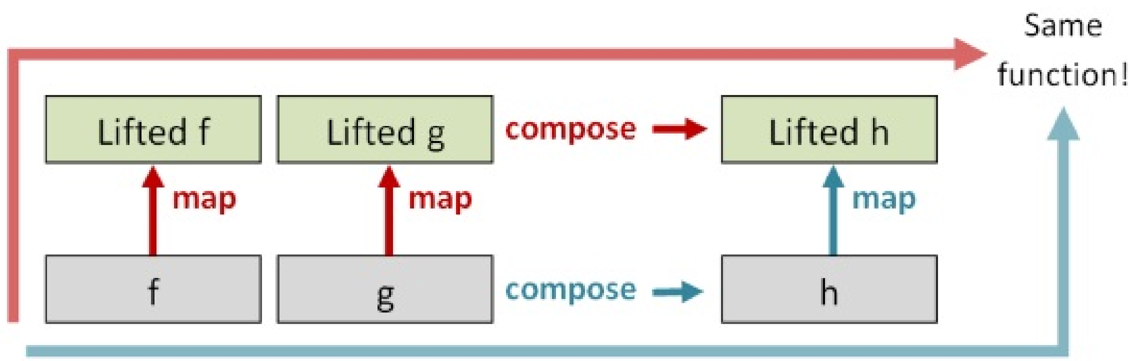
In my post on property based testing I showed how a correct implementation of a function can be defined and tested using general properties rather than specific examples.

This is true for `map` as well. The implementation will vary with the specific elevated world, but in all cases, there are certain properties that the implementation should satisfy to avoid strange behavior.

First, if you take the `id` function in the normal world, and you lift it into the elevated world with `map`, the new function must be the *same* as the `id` function in the elevated world.



Next, if you take two functions `f` and `g` in the normal world, and you compose them (into `h`, say), and then lift the resulting function using `map`, the resulting function should be the *same* as if you lifted `f` and `g` into the elevated world *first*, and then composed them there afterwards.



These two properties are the so-called "Functor Laws", and a **Functor** (in the programming sense) is defined as a generic data type -- `E<T>` in our case -- plus a `map` function that obeys the functor laws.

NOTE: "Functor" is a confusing word. There is "functor" in the category theory sense, and there is "functor" in the programming sense (as defined above). There are also things called "functors" defined in libraries, such as the `Functor` type class in Haskell, and the `Functor` trait in Scalaz, not to mention functors in SML and OCaml (and C++), which are different yet again!

Conseqently, I prefer to talk about "mappable" worlds. In practical programming, you will almost never run into a elevated world that does not support being mapped over somehow.

Variants of map

There are some variants of map that are common:

- **Const map.** A const or "replace-by" map replaces all values with a constant rather than the output of a function. In some cases, a specialized function like this can allow for a more efficient implementation.
- **Maps that work with cross-world functions.** The map function `a->b` lives entirely in the normal world. But what if the function you want to map with does not return something in the normal world, but a value in another, different, enhanced world? We'll see how to address this challenge in a later post.

The `return` function

Common Names: `return` , `pure` , `unit` , `yield` , `point`

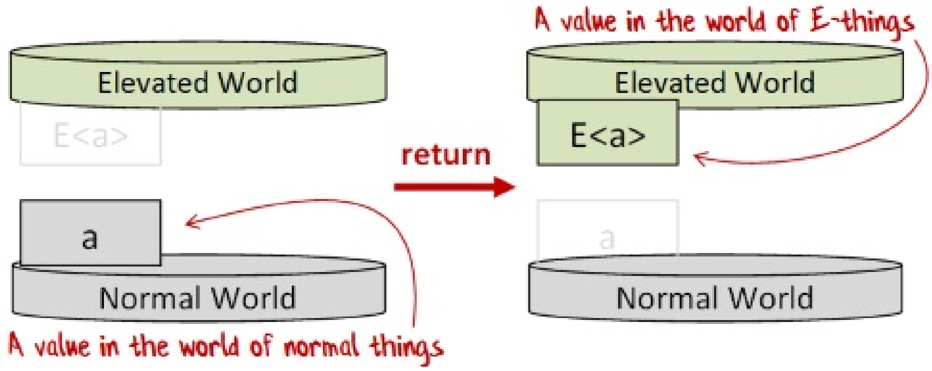
Common Operators: None

What it does: Lifts a single value into the elevated world

Signature: `a -> E<a>`

Description

"return" (also known as "unit" or "pure") simply creates a elevated value from a normal value.



This function goes by many names, but I'm going to be consistent and call it `return` as that is the common term for it in F#, and is the term used in computation expressions.

NOTE: I'm ignoring the difference between `pure` and `return` , because type classes are not the focus of this post.

Implementation examples

Here are two examples of `return` implementations in F#:

```
// A value lifted to the world of Options
let returnOption x = Some x
// has type : 'a -> 'a option

// A value lifted to the world of Lists
let returnList x  = [x]
// has type : 'a -> 'a list
```

Obviously, we don't need to define special functions like this for options and lists. Again, I've just done it to show what `return` might look for some common types.

The `apply` function

Common Names: `apply` , `ap`

Common Operators: `<*>`

What it does: Unpacks a function wrapped inside a elevated value into a lifted function `E<a> -> E<b>`

Signature: `E<(a->b)> -> E<a> -> E<b>`

Description

"apply" unpacks a function wrapped inside a elevated value (`E<(a->b)>`) into a lifted function `E<a> -> E<b>`



This might seem unimportant, but is actually very valuable, as it allows you to lift a multi-parameter function in the normal world into a multi-parameter function in the elevated world, as we'll see shortly.

Alternative interpretation

An alternative interpretation of `apply` is that it is a *two* parameter function that takes a elevated value (`E<a>`) and a elevated function (`E<(a->b)>`), and returns a new elevated value (`E<b>`) generated by applying the function `a->b` to the internal elements of `E<a>` .

For example, if you have a one-parameter function (`E<(a->b)>`), you can apply it to a single elevated parameter to get the output as another elevated value.



If you have a two-parameter function (`E<(a->b->c)>`), you can use `apply` twice in succession with two elevated parameters to get the elevated output.



You can continue to use this technique to work with as many parameters as you wish.

Implementation examples

Here are some examples of defining `apply` for two different types in F#:

```
module Option =

    // The apply function for Options
    let apply fOpt xOpt =
        match fOpt,xOpt with
        | Some f, Some x -> Some (f x)
        | _ -> None

module List =

    // The apply function for lists
    // [f;g] apply [x;y] becomes [f x; f y; g x; g y]
    let apply (fList: ('a->'b) list) (xList: 'a list)  =
        [ for f in fList do
          for x in xList do
              yield f x ]
```

In this case, rather than have names like `applyOption` and `applyList`, I have given the functions the same name but put them in a per-type module.

Note that in the `List.apply` implementation, each function in the first list is applied to each value in the second list, resulting in a "cross-product" style result. That is, the list of functions `[f; g]` applied to the list of values `[x; y]` becomes the four-element list `[f x; f y; g x; g y]`. We'll see shortly that this is not the only way to do it.

Also, of course, I'm cheating by building this implementation on a `for..in..do` loop -- functionality that already exists!

I did this for clarity in showing how `apply` works. It's easy enough to create a "from scratch" recursive implementation, (though it is not so easy to create one that is properly tail-recursive!) but I want to focus on the concepts not on the implementation for now.

Infix version of apply

Using the `apply` function as it stands can be awkward, so it is common to create an infix version, typically called `<*>`. With this in place you can write code like this:

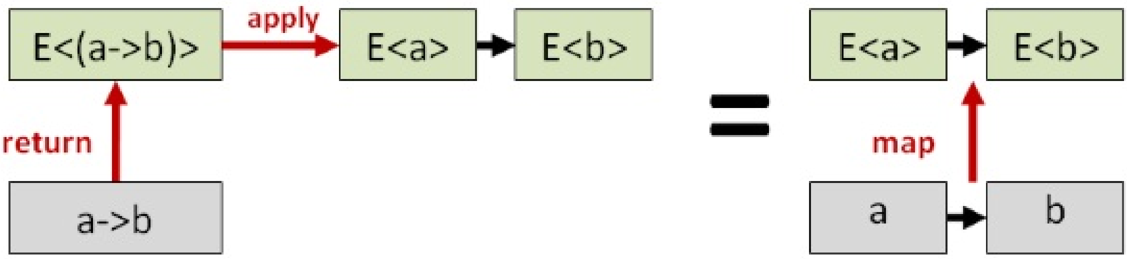
```
let resultOption =
    let (<*>) = Option.apply
    (Some add) <*> (Some 2) <*> (Some 3)
// resultOption = Some 5

let resultList =
    let (<*>) = List.apply
    [add] <*> [1;2] <*> [10;20]
// resultList = [11; 21; 12; 22]
```

Apply vs. Map

The combination of `apply` and `return` is considered "more powerful" than `map`, because if you have `apply` and `return`, you can construct `map` from them, but not vice versa.

Here's how it works: to construct a lifted function from a normal function, just use `return` on the normal function and then `apply`. This gives you the same result as if you had simply done `map` in the first place.



This trick also means that our infix notation can be simplified a little. The initial `return` then `apply` can be replaced with `map`, and we so typically create an infix operator for `map` as well, such as `<!>` in F#.

```
let resultOption2 =
    let (<!>) = Option.map
    let (<*>) = Option.apply

    add <!> (Some 2) <*> (Some 3)
// resultOption2 = Some 5

let resultList2 =
    let (<!>) = List.map
    let (<*>) = List.apply

    add <!> [1;2] <*> [10;20]
// resultList2 = [11; 21; 12; 22]
```

This makes the code look more like using the function normally. That is, instead of the normal `add x y`, we can use the similar looking `add <!> x <*> y`, but now `x` and `y` can be elevated values rather than normal values. Some people have even called this style "overloaded whitespace"!

Here's one more for fun:

```
let batman =
    let (<!>) = List.map
    let (<*>) = List.apply

    // string concatenation using +
    (+) <!> ["bam"; "kapow"; "zap"] <*> ["!"; "!!"]

// result =
// ["bam!"; "bam!!"; "kapow!"; "kapow!!"; "zap!"; "zap!!"]
```

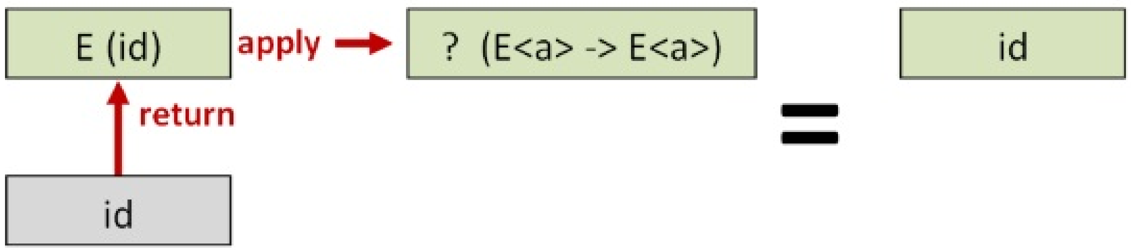
The properties of a correct apply/return implementation

As with `map`, a correct implementation of the `apply` / `return` pair should have some properties that are true no matter what elevated world we are working with.

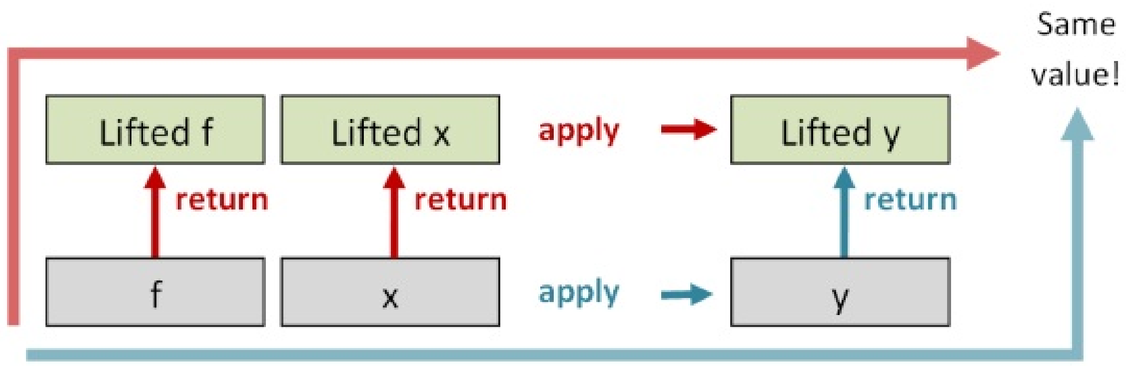
There are four so-called "Applicative Laws", and an **Applicative Functor** (in the programming sense) is defined as a generic data type constructor -- `E<T>` -- plus a pair of functions (`apply` and `return`) that obey the applicative laws.

Just as with the laws for `map`, these laws are quite sensible. I'll show two of them.

The first law says that if you take the `id` function in the normal world, and you lift it into the elevated world with `return`, and then you do `apply`, the new function, which is of type `E<a> -> E<a>`, should be the same as the `id` function in the elevated world.



The second law says that if you take a function `f` and a value `x` in the normal world, and you apply `f` to `x` to get a result (`y`, say), and then lift the result using `return`, the resulting value should be the same as if you lifted `f` and `x` into the elevated world *first*, and then applied them there afterwards.



The other two laws are not so easily diagrammed, so I won't document them here, but together the laws ensure that any implementation is sensible.

The `liftN` family of functions

Common Names: `lift2` , `lift3` , `lift4` and similar

Common Operators: None

What it does: Combines two (or three, or four) elevated values using a specified function

Signature:
lift2: `(a->b->c) -> E<a> -> E<b> -> E<c>` ,
lift3: `(a->b->c->d) -> E<a> -> E<b> -> E<c> -> E<d>` ,
etc.

Description

The `apply` and `return` functions can be used to define a series of helper functions `liftN` (`lift2` , `lift3` , `lift4` , etc) that take a normal function with N parameters (where N=2,3,4, etc) and transform it to a corresponding elevated function.

Note that `lift1` is just `map` , and so it is not usually defined as a separate function.

Here's what an implementation might look like:

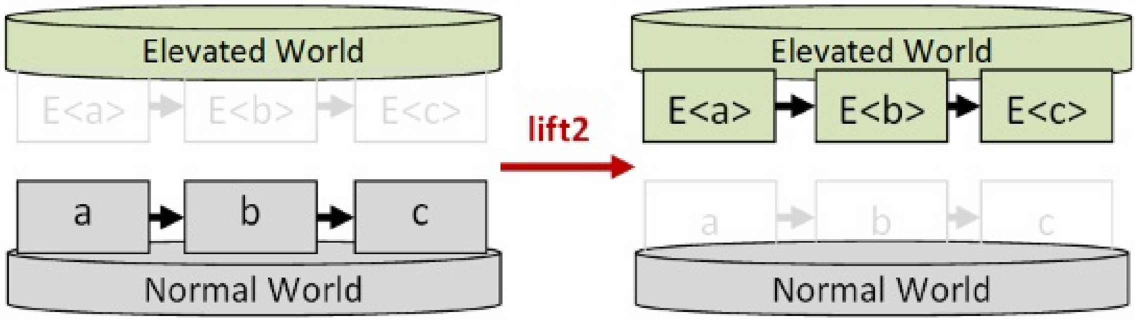
```
module Option =
    let (<*>) = apply
    let (<!>) = Option.map

    let lift2 f x y =
        f <!> x <*> y

    let lift3 f x y z =
        f <!> x <*> y <*> z

    let lift4 f x y z w =
        f <!> x <*> y <*> z <*> w
```

Here's a visual representation of `lift2` :



The `lift` series of functions can be used to make code a bit more readable because, by using one of the pre-made `lift` functions, we can avoid the `<*>` syntax.

First, here's an example of lifting a two-parameter function:

```
// define a two-parameter function to test with
let addPair x y = x + y

// lift a two-param function
let addPairOpt = Option.lift2 addPair

// call as normal
addPairOpt (Some 1) (Some 2)
// result => Some 3
```

And here's an example of lifting a three-parameter function:

```
// define a three-parameter function to test with
let addTriple x y z = x + y + z

// lift a three-param function
let addTripleOpt = Option.lift3 addTriple

// call as normal
addTripleOpt (Some 1) (Some 2) (Some 3)
// result => Some 6
```

Interpreting "lift2" as a "combiner"

There is an alternative interpretation of `apply` as a "combiner" of elevated values, rather than as function application.

For example, when using `lift2`, the first parameter is a parameter specifying how to combine the values.

Here's an example where the same values are combined in two different ways: first with addition, then with multiplication.

```
Option.lift2 (+) (Some 2) (Some 3)   // Some 5
Option.lift2 (*) (Some 2) (Some 3)   // Some 6
```

Going further, can we eliminate the need for this first function parameter and have a *generic* way of combining the values?

Why, yes we can! We can just use a tuple constructor to combine the values. When we do this we are combining the values without making any decision about how they will be used yet.

Here's what it looks like in a diagram:



and here's how you might implement it for options and lists:

```
// define a tuple creation function
let tuple x y = x,y

// create a generic combiner of options
// with the tuple constructor baked in
let combineOpt x y = Option.lift2 tuple x y

// create a generic combiner of lists
// with the tuple constructor baked in
let combineList x y = List.lift2 tuple x y
```

Let's see what happens when we use the combiners:

```
combineOpt (Some 1) (Some 2)
// Result => Some (1, 2)

combineList [1;2] [100;200]
// Result => [(1, 100); (1, 200); (2, 100); (2, 200)]
```

Now that we have an elevated tuple, we can work with the pair in any way we want, we just need to use `map` to do the actual combining.

Want to add the values? Just use `+` in the `map` function:

```
combineOpt (Some 2) (Some 3)
|> Option.map (fun (x,y) -> x + y)
// Result => // Some 5

combineList [1;2] [100;200]
|> List.map (fun (x,y) -> x + y)
// Result => [101; 201; 102; 202]
```

Want to multiply the values? Just use `*` in the `map` function:

```
combineOpt (Some 2) (Some 3)
|> Option.map (fun (x,y) -> x * y)
// Result => Some 6

combineList [1;2] [100;200]
|> List.map (fun (x,y) -> x * y)
// Result => [100; 200; 200; 400]
```

And so on. Obviously, real-world uses would be somewhat more interesting.

Defining `apply` in terms of `lift2`

Interestingly, the `lift2` function above can be actually used as an alternative basis for defining `apply`.

That is, we can define `apply` in terms of the `lift2` function by setting the combining function to be just function application.

Here's a demonstration of how this works for `Option`:

```
module Option =

    /// define lift2 from scratch
    let lift2 f xOpt yOpt =
        match xOpt,yOpt with
        | Some x,Some y -> Some (f x y)
        | _ -> None

    /// define apply in terms of lift2
    let apply fOpt xOpt =
        lift2 (fun f x -> f x) fOpt xOpt
```

This alternative approach is worth knowing about because for some types it's easier to define `lift2` than `apply`.

Combining missing or bad data

Notice that in all the combiners we've looked at, if one of the elevated values is "missing" or "bad" somehow, then the overall result is also bad.

For example, with `combineList`, if one of the parameters is an empty list, the result is also an empty list, and with `combineOpt`, if one of the parameters is `None`, the result is also `None`.

```
combineOpt (Some 2) None
|> Option.map (fun (x,y) -> x + y)
// Result => None

combineList [1;2] []
|> List.map (fun (x,y) -> x * y)
// Result => Empty list
```

It's possible to create an alternative kind of combiner that ignores missing or bad values, just as adding "0" to a number is ignored. For more information, see my post on "Monoids without tears".

One sided combiners `<*` and `*>`

In some cases you might have two elevated values, and want to discard the value from one side or the other.

Here's an example for lists:

```
let ( <* ) x y =
    List.lift2 (fun left right -> left) x y

let ( *> ) x y =
    List.lift2 (fun left right -> right) x y
```

We can then combine a 2-element list and a 3-element list to get a 6-element list as expected, but the contents come from only one side or the other.

```
[1;2] <* [3;4;5]   // [1; 1; 1; 2; 2; 2]
[1;2] *> [3;4;5]   // [3; 4; 5; 3; 4; 5]
```

We can turn this into a feature! We can replicate a value N times by crossing it with `[1..n]`.

```
let repeat n pattern =
    [1..n] *> pattern

let replicate n x =
    [1..n] *> [x]

repeat 3 ["a";"b"]
// ["a"; "b"; "a"; "b"; "a"; "b"]

replicate 5 "A"
// ["A"; "A"; "A"; "A"; "A"]
```

Of course, this is by no means an efficient way to replicate a value, but it does show that starting with just the two functions `apply` and `return`, you can build up some quite complex behavior.

On a more practical note though, why might this "throwing away data" be useful? Well in many cases, we might not want the values, but we *do* want the effects.

For example, in a parser, you might see code like this:

```
let readQuotedString =
   readQuoteChar *> readNonQuoteChars <* readQuoteChar
```

In this snippet, `readQuoteChar` means "match and read a quote character from the input stream" and `readNonQuoteChars` means "read a series of non-quote characters from the input stream".

When we are parsing a quoted string we want ensure the input stream that contains the quote character is read, but we don't care about the quote characters themselves, just the inner content.

Hence the use of `*>` to ignore the leading quote and `<*` to ignore the trailing quote.

The `zip` function and ZipList world

Common Names: `zip`, `zipWith`, `map2`

Common Operators: `<*>` (in the context of ZipList world)

What it does: Combines two lists (or other enumerables) using a specified function

Signature: `E<(a->b->c)> -> E<a> -> E<b> -> E<c>` where `E` is a list or other enumerable type, or `E<a> -> E<b> -> E<a,b>` for the tuple-combined version.

Description

Some data types might have more than one valid implementation of `apply`. For example, there is another possible implementation of `apply` for lists, commonly called `ZipList` or some variant of that.

In this implementation, the corresponding elements in each list are processed at the same time, and then both lists are shifted to get the next element. That is, the list of functions `[f; g]` applied to the list of values `[x; y]` becomes the two-element list `[f x; g y]`

```
// alternate "zip" implementation
// [f;g] apply [x;y] becomes [f x; g y]
let rec zipList fList xList  =
    match fList,xList with
    | [],_
    | _,[] ->
        // either side empty, then done
        []
    | (f::fTail),(x::xTail) ->
        // new head + new tail
        (f x) :: (zipList fTail xTail)
// has type : ('a -> 'b) -> 'a list -> 'b list
```

WARNING: This implementation is just for demonstration. It's not tail-recursive, so don't use it for large lists!

If the lists are of different lengths, some implementations throw an exception (as the F# library functions `List.map2` and `List.zip` do), while others silently ignore the extra data (as the implementation above does).

Ok, let's see it in use:

```
let add10 x = x + 10
let add20 x = x + 20
let add30 x = x + 30

let result =
    let (<*>) = zipList
    [add10; add20; add30] <*> [1; 2; 3]
// result => [11; 22; 33]
```

Note that the result is `[11; 22; 33]` -- only three elements. If we had used the standard `List.apply`, there would have been nine elements.

Interpreting "zip" as a "combiner"

We saw above that `List.apply`, or rather `List.lift2`, could be intepreted as a combiner. Similarly, so can `zipList`.

```
let add x y = x + y

let resultAdd =
    let (<*>) = zipList
    [add;add] <*> [1;2] <*> [10;20]
// resultAdd = [11; 22]
// [ (add 1 10); (add 2 20) ]
```

Note that we can't just have *one* `add` function in the first list -- we have to have one `add` for every element in the second and third lists!

That could get annoying, so often, a "tupled" version of `zip` is used, whereby you don't specify a combining function at all, and just get back a list of tuples instead, which you can then process later using `map`. This is the same approach as was used in the `combine` functions discussed above, but for `zipList`.

ZipList world

In standard List world, there is an `apply` and a `return`. But with our different version of `apply` we can create a different version of List world called ZipList world.

ZipList world is quite different from the standard List world.

In ZipList world, the `apply` function is implemented as above. But more interestingly, ZipList world has a *completely different* implementation of `return` compared with standard List world. In the standard List world, `return` is just a list with a single element, but for ZipList world, it has to be an infinitely repeated value!

In a non-lazy language like F#, we can't do this, but if we replace `List` with `Seq` (aka `IEnumerable`) then we *can* create an infinitely repeated value, as shown below:

```
module ZipSeq =

    // define "return" for ZipSeqWorld
    let retn x = Seq.initInfinite (fun _ -> x)

    // define "apply" for ZipSeqWorld
    // (where we can define apply in terms of "lift2", aka "map2")
    let apply fSeq xSeq  =
        Seq.map2 (fun f x -> f x)  fSeq xSeq
    // has type : ('a -> 'b) seq -> 'a seq -> 'b seq

    // define a sequence that is a combination of two others
    let triangularNumbers =
        let (<*>) = apply

        let addAndDivideByTwo x y = (x + y) / 2
        let numbers = Seq.initInfinite id
        let squareNumbers = Seq.initInfinite (fun i -> i * i)
        (retn addAndDivideByTwo) <*> numbers <*> squareNumbers

    // evaulate first 10 elements
    // and display result
    triangularNumbers |> Seq.take 10 |> List.ofSeq |> printfn "%A"
    // Result =>
    // [0; 1; 3; 6; 10; 15; 21; 28; 36; 45]
```

This example demonstrates that an elevated world is *not* just a data type (like the List type) but consists of the datatype *and* the functions that work with it. In this particular case, "List world" and "ZipList world" share the same data type but have quite different environments.

What types support `map` and `apply` and `return`?

So far we have defined all these useful functions in an abstract way. But how easy is it to find real types that have implementations of them, including all the various laws?

The answer is: very easy! In fact *almost all* types support these set of functions. You'd be hard-pressed to find a useful type that didn't.

That means that `map` and `apply` and `return` are available (or can be easily implemented) for standard types such as `Option`, `List`, `Seq`, `Async`, etc., and also any types you are likely to define yourself.

Summary

In this post, I described three core functions for lifting simple "normal" values to elevated worlds: `map`, `return`, and `apply`, plus some derived functions like `liftN` and `zip`.

In practice however, things are not that simple. We frequently have to work with functions that cross between the worlds. Their input is in the normal world but their output is in the elevated world.

In the next post we'll show how these world-crossing functions can be lifted to the elevated world as well.

Understanding bind

This post is the second in a series. In the previous post, I described some of the core functions for lifting a value from a normal world to an elevated world.

In this post, we'll look at "world-crossing" functions, and how they can be tamed with the `bind` function.

Series contents

Here's a list of shortcuts to the various functions mentioned in this series:

- **Part 1: Lifting to the elevated world**
  - The `map` function
  - The `return` function
  - The `apply` function
  - The `liftN` family of functions
  - The `zip` function and ZipList world
- **Part 2: How to compose world-crossing functions**
  - The `bind` function
  - List is not a monad. Option is not a monad.
- **Part 3: Using the core functions in practice**
  - Independent and dependent data
  - Example: Validation using applicative style and monadic style
  - Lifting to a consistent world
  - Kleisli world
- **Part 4: Mixing lists and elevated values**
  - Mixing lists and elevated values
  - The `traverse` / `MapM` function
  - The `sequence` function
  - "Sequence" as a recipe for ad-hoc implementations
  - Readability vs. performance
  - Dude, where's my `filter` ?
- **Part 5: A real-world example that uses all the techniques**
  - Example: Downloading and processing a list of websites
  - Treating two worlds as one
- **Part 6: Designing your own elevated world**
  - Designing your own elevated world

- Filtering out failures
- The Reader monad
- **Part 7: Summary**
  - List of operators mentioned
  - Further reading

Part 2: How to compose world-crossing functions

The `bind` function

Common Names: `bind`, `flatMap`, `andThen`, `collect`, `SelectMany`

Common Operators: `>>=` (left to right), `=<<` (right to left)

What it does: Allows you to compose world-crossing ("monadic") functions

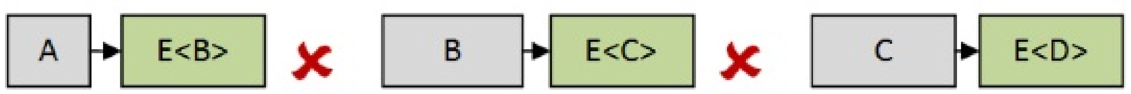
Signature: `(a->E<b>) -> E<a> -> E<b>`. Alternatively with the parameters reversed: `E<a> -> (a->E<b>) -> E<b>`

Description

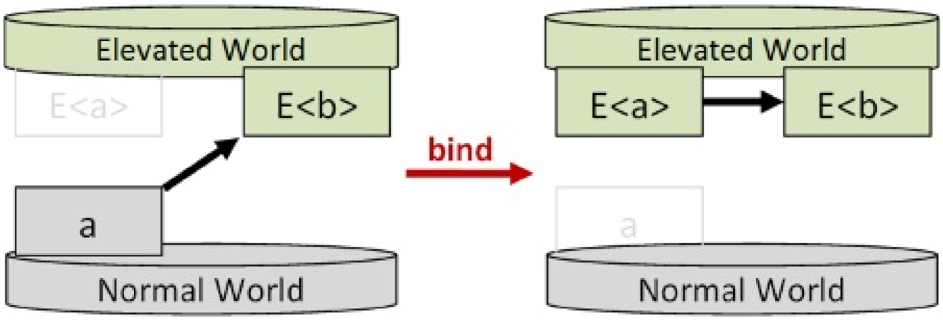
We frequently have to deal with functions that cross between the normal world and the elevated world.

For example: a function that parses a `string` to an `int` might return an `Option<int>` rather than a normal `int`, a function that reads lines from a file might return `IEnumerable<string>`, a function that fetches a web page might return `Async<string>`, and so on.

These kinds of "world-crossing" functions are recognizable by their signature `a -> E<b>`; their input is in the normal world but their output is in the elevated world. Unfortunately, this means that these kinds of functions cannot be linked together using standard composition.

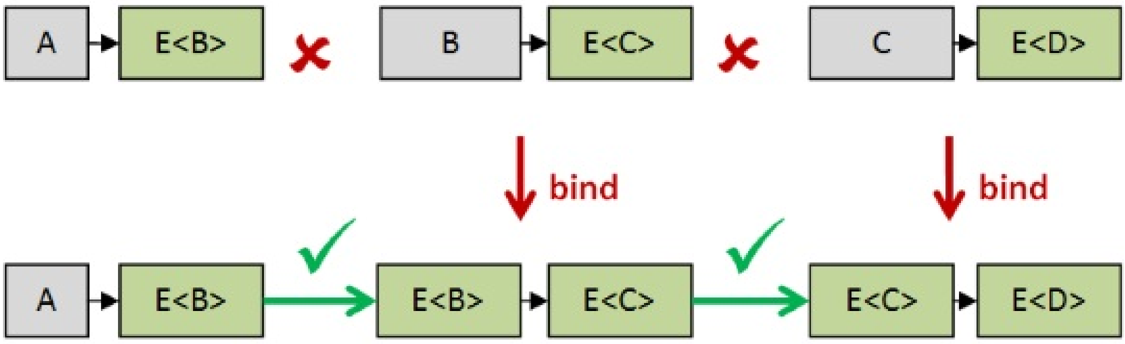


What "bind" does is transform a world-crossing function (commonly known as a "monadic function") into a lifted function `E<a> -> E<b>`.



The benefit of doing this is that the resulting lifted functions live purely in the elevated world, and so can be combined easily by composition.

For example, a function of type `a -> E<b>` cannot be directly composed with a function of type `b -> E<c>`, but after `bind` is used, the second function becomes of type `E<b> -> E<c>`, which *can* be composed.



In this way, `bind` lets us chain together any number of monadic functions.

Alternative interpretation

An alternative interpretation of `bind` is that it is a *two* parameter function that takes a elevated value (`E<a>`) and a "monadic function" (`a -> E<b>`), and returns a new elevated value (`E<b>`) generated by "unwrapping" the value inside the input, and running the function `a -> E<b>` against it. Of course, the "unwrapping" metaphor does not work for every elevated world, but still it can often be useful to think of it this way.



Implementation examples

Here are some examples of defining `bind` for two different types in F#:

```
module Option =

    // The bind function for Options
    let bind f xOpt =
        match xOpt with
        | Some x -> f x
        | _ -> None
    // has type : ('a -> 'b option) -> 'a option -> 'b option

module List =

    // The bind function for lists
    let bindList (f: 'a->'b list) (xList: 'a list)  =
        [ for x in xList do
          for y in f x do
              yield y ]
    // has type : ('a -> 'b list) -> 'a list -> 'b list
```

Notes:

- Of course, in these two particular cases, the functions already exist in F#, called `Option.bind` and `List.collect`.
- For `List.bind` I'm cheating again and using `for..in..do`, but I think that this particular implementation shows clearly how bind works with lists. There is a purer recursive implementation, but I won't show that here.

Usage example

As explained at the beginning of this section, `bind` can be used to compose cross-world functions. Let's see how this works in practice with a simple example.

First let's say we have a function that parses certain `string`s into `int`s. Here's a very simple implementation:

```
let parseInt str =
    match str with
    | "-1" -> Some -1
    | "0" -> Some 0
    | "1" -> Some 1
    | "2" -> Some 2
    // etc
    | _ -> None

// signature is string -> int option
```

Sometimes it returns a int, sometimes not. So the signature is `string -> int option` -- a cross-world function.

And let's say we have another function that takes an `int` as input and returns a `OrderQty` type:

```
type OrderQty = OrderQty of int

let toOrderQty qty =
    if qty >= 1 then
        Some (OrderQty qty)
    else
        // only positive numbers allowed
        None

// signature is int -> OrderQty option
```

Again, it might not return an `OrderQty` if the input is not positive. The signature is therefore `int -> OrderQty option` -- another cross-world function.

Now, how can we create a function that starts with an string and returns an `OrderQty` in one step?

The output of `parseInt` cannot be fed directly into `toOrderQty`, so this is where `bind` comes to the rescue!

Doing `Option.bind toOrderQty` lifts it to a `int option -> OrderQty option` function and so the output of `parseInt` can be used as input, just as we need.

```
let parseOrderQty str =
    parseInt str
    |> Option.bind toOrderQty
// signature is string -> OrderQty option
```

The signature of our new `parseOrderQty` is `string -> OrderQty option`, yet another cross-world function. So if we want to do something with the `OrderQty` that is output we may well have to use `bind` again on the next function in the chain.

Infix version of bind

As with `apply`, using the named `bind` function can be awkward, so it is common to create an infix version, typically called `>>=` (for left to right data flow) or `=<<` (for right to left data flow) .

With this in place you can write an alternative version of `parseOrderQty` like this:

```
let parseOrderQty_alt str =
    str |> parseInt >>= toOrderQty
```

You can see that `>>=` performs the same kind of role as pipe (`|>`) does, except that it works to pipe "elevated" values into cross-world functions.

Bind as a "programmable semicolon"

Bind can be used to chain any number of functions or expressions together, so you often see code looking something like this:

```
expression1 >>=
expression2 >>=
expression3 >>=
expression4
```

This is not too different from how an imperative program might look if you replace the `>>=` with a `;` :

```
statement1;
statement2;
statement3;
statement4;
```

Because of this, `bind` is sometimes called a "programmable semicolon".

Language support for bind/return

Most functional programming languages have some kind of syntax support for `bind` that lets you avoid having to write a series of continuations or use explicit binds.

In F# it is (one component) of computation expressions, so the following explicit chaining of `bind` :

```
initialExpression >>= (fun x ->
expressionUsingX  >>= (fun y ->
expressionUsingY  >>= (fun z ->
x+y+z )))             // return
```

becomes implicit, using `let!` syntax:

```
elevated {
    let! x = initialExpression
    let! y = expressionUsingX x
    let! z = expressionUsingY y
    return x+y+z }
```

In Haskell, the equivalent is the "do notation":

```
do
    x <- initialExpression
    y <- expressionUsingX x
    z <- expressionUsingY y
    return x+y+z
```

And in Scala, the equivalent is the "for comprehension":

```
for {
    x <- initialExpression
    y <- expressionUsingX(x)
    z <- expressionUsingY(y)
} yield {
    x+y+z
}
```

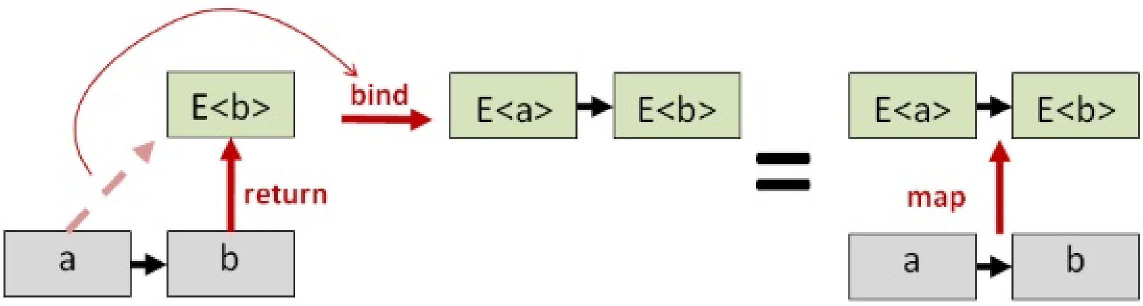
It's important to emphasize that you do not *have* to use the special syntax when using bind/return. You can always use `bind` or `>>=` in the same way as any other function.

Bind vs. Apply vs. Map

The combination of `bind` and `return` are considered even more powerful than `apply` and `return`, because if you have `bind` and `return`, you can construct `map` and `apply` from them, but not vice versa.

Here's how bind can be used to emulate `map`, for example:

- First, you construct a world-crossing function from a normal function by applying `return` to the output.
- Next, convert this world-crossing function into a lifted function using `bind`. This gives you the same result as if you had simply done `map` in the first place.



Similarly, `bind` can emulate `apply`. Here is how `map` and `apply` can be defined using `bind` and `return` for Options in F#:

```
// map defined in terms of bind and return (Some)
let map f =
    Option.bind (f >> Some)

// apply defined in terms of bind and return (Some)
let apply fOpt xOpt =
    fOpt |> Option.bind (fun f ->
        let map = Option.bind (f >> Some)
        map xOpt)
```

At this point, people often ask "why should I use `apply` instead of `bind` when `bind` is more powerful?"

The answer is that just because `apply` *can* be emulated by `bind`, doesn't mean it *should* be. For example, it is possible to implement `apply` in a way that cannot be emulated by a `bind` implementation.

In fact, using `apply` ("applicative style") or `bind` ("monadic style") can have a profound effect on how your program works! We'll discuss these two approaches in more detail in part 3 of this post.

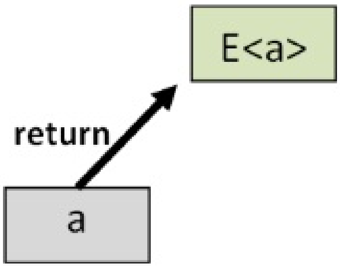
The properties of a correct bind/return implementation

As with `map`, and as with `apply` / `return`, a correct implementation of the `bind` / `return` pair should have some properties that are true no matter what elevated world we are working with.

There are three so-called "Monad Laws", and one way of defining a **Monad** (in the programming sense) is to say that it consists of three things: a generic type constructor `E<T>` plus a pair of functions (`bind` and `return`) that obey the monad laws. This is not the only way to define a monad, and mathematicians typically use a slightly different definition, but this one is most useful to programmers.

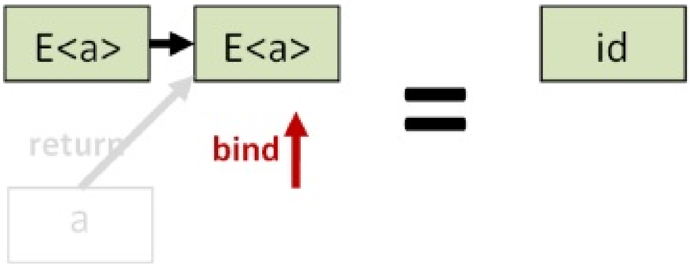
Just as with the the Functor and Applicative laws we saw earlier, these laws are quite sensible.

First, note that `return` function is itself a cross-world function:

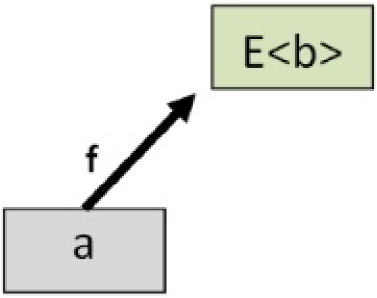


That means that we can use `bind` to lift it into a function in the elevated world. And what does this lifted function do? Hopefully, nothing! It should just return its input.

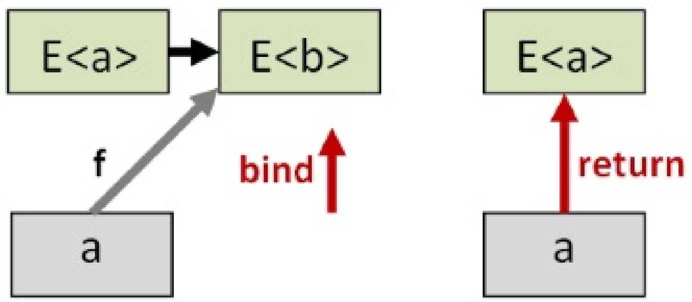
So that is exactly the first monad law: it says that this lifted function must be the same as the `id` function in the elevated world.



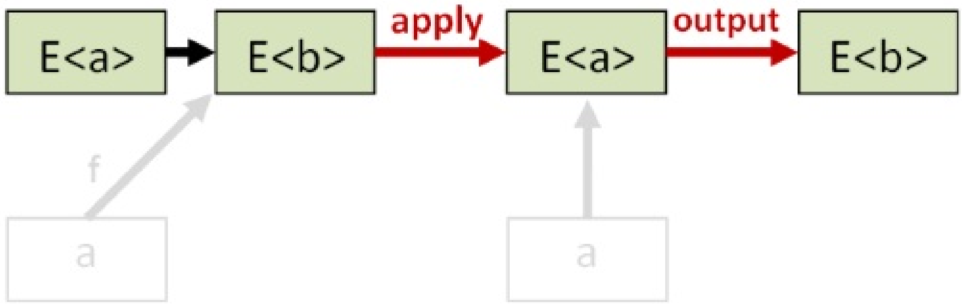
The second law is similar but with `bind` and `return` reversed. Say that we have a normal value `a` and cross-world function `f` that turns an `a` into a `E<b>`.



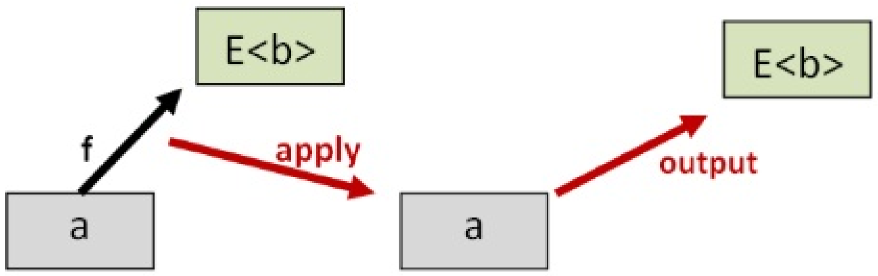
Let's lift both of them to the elevated world, using `bind` on `f` and `return` on `a`.



Now if we apply the elevated version of `f` to the elevated verson of `a` we get some value `E<b>`.



On the other hand if we apply the normal version of `f` to the normal verson of `a` we *also* get some value `E<b>`.



The second monad law says that these two elevated values (`E<b>`) should be the same. In other words, all this binding and returning should not distort the data.

The third monad law is about associativity.

In the normal world, function composition is associative. For example, we could pipe a value into a function `f` and then take that result and pipe it into another function `g`. Alternatively, we can compose `f` and `g` first into a single function and then pipe `a` into it.

```
let groupFromTheLeft = (a |> f) |> g
let groupFromTheRight = a |> (f >> g)
```

In the normal world, we expect both of these alternatives to give the same answer.

The third monad law says that, after using `bind` and `return`, the grouping doesn't matter either. The two examples below correspond to the examples above:

```
let groupFromTheLeft = (a >>= f) >>= g
let groupFromTheRight = a >>= (fun x -> f x >>= g)
```

And again, we expect both of these to give the same answer.

List is not a monad. Option is not a monad.

If you look at the definition above, a monad has a type constructor (a.k.a "generic type") *and* two functions *and* a set of properties that must be satisfied.

The `List` data type is therefore just one component of a monad, as is the `Option` data type. `List` and `Option`, by themselves, are not monads.

It might be better to think of a monad as a *transformation*, so that the "List monad" is the transformation that converts the normal world to the elevated "List world", and the "Option monad" is the transformation that converts the normal world to the elevated "Option world".

I think this is where a lot of the confusion comes in. The word "List" can mean many different things:

1. A concrete type or data structure such as `List<int>`.
2. A type constructor (generic type): `List<T>`.
3. A type constructor and some operations, such as a `List` class or module.
4. A type constructor and some operations and the operations satisfy the monad laws.

Only the last one is a monad! The other meanings are valid but contribute to the confusion.

Also the last two cases are hard to tell apart by looking at the code. Unfortunately, there have been cases where implementations did not satisfy the monad laws. Just because it's a "Monad" doesn't mean that it's a monad.

Personally, I try to avoid using the word "monad" on this site and focus on the `bind` function instead, as part of a toolkit of functions for solving problems rather than an abstract concept.

So don't ask: Do I have a monad?

Do ask: Do I have useful bind and return functions? And are they implemented correctly?

Summary

We now have a set of four core functions: `map`, `return`, `apply`, and `bind`, and I hope that you are clear on what each one does.

But there are some questions that have not been addressed yet, such as "why should I choose `apply` instead of `bind`?", or "how can I deal with multiple elevated worlds at the same time?"

In the next post we'll address these questions and demonstrate how to use the toolset with a series of practical examples.

UPDATE: Fixed error in monad laws pointed out by @joseanpg. Thanks!

Using the core functions in practice

This post is the third in a series. In the previous two posts, I described some of the core functions for dealing with generic data types: `map` , `apply` , `bind` , and so on.

In this post, I'll show how to use these functions in practice, and will explain the difference between the so-called "applicative" and "monadic" styles.

Series contents

Here's a list of shortcuts to the various functions mentioned in this series:

- **Part 1: Lifting to the elevated world**
  - The `map` function
  - The `return` function
  - The `apply` function
  - The `liftN` family of functions
  - The `zip` function and ZipList world
- **Part 2: How to compose world-crossing functions**
  - The `bind` function
  - List is not a monad. Option is not a monad.
- **Part 3: Using the core functions in practice**
  - Independent and dependent data
  - Example: Validation using applicative style and monadic style
  - Lifting to a consistent world
  - Kleisli world
- **Part 4: Mixing lists and elevated values**
  - Mixing lists and elevated values
  - The `traverse` / `MapM` function
  - The `sequence` function
  - "Sequence" as a recipe for ad-hoc implementations
  - Readability vs. performance
  - Dude, where's my `filter` ?
- **Part 5: A real-world example that uses all the techniques**
  - Example: Downloading and processing a list of websites
  - Treating two worlds as one
- **Part 6: Designing your own elevated world**
  - Designing your own elevated world

- Filtering out failures
- The Reader monad
- **Part 7: Summary**
  - List of operators mentioned
  - Further reading

Part 3: Using the core functions in practice

Now that we have the basic tools for lifting normal values to elevated values and working with cross-world functions, it's time to start working with them!

In this section, we'll look at some examples how these functions are actually used.

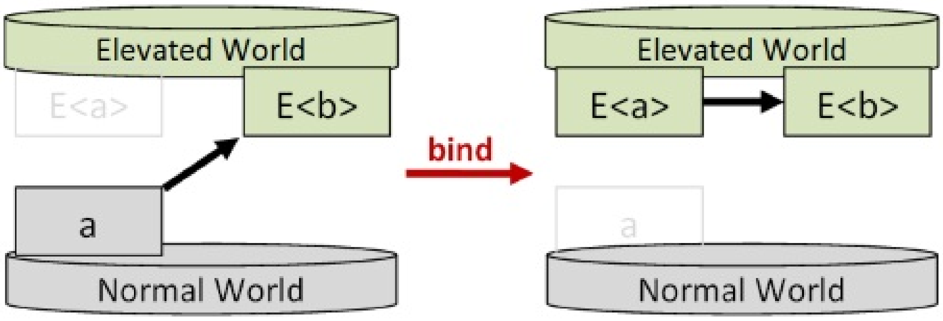
Independent vs. dependent data

I briefly mentioned earlier that there is a important difference between using `apply` and `bind`. Let's go into this now.

When using `apply`, you can see that each parameter (`E<a>`, `E<b>`) is completely independent of the other. The value of `E<b>` does not depend on what `E<a>` is.



On the other hand, when using `bind`, the value of `E<b>` *does* depend on what `E<a>` is.



The distinction between working with independent values or dependent values leads to two different styles:

- The so-called "applicative" style uses functions such as `apply`, `lift`, and `combine` where each elevated value is independent.
- The so-called "monadic" style uses functions such as `bind` to chain together functions

that are dependent on a previous value.

What does that mean in practice? Well, let's look at an example where you could choose from both approaches.

Say that you have to download data from three websites and combine them. And say that we have an action, say `GetURL`, that gets the data from a website on demand.

Now you have a choice:

- **Do you want to fetch all the URLs in parallel?** If so, treat the `GetURL` s as independent data and use the applicative style.
- **Do you want to fetch each URL one at a time, and skip the next in line if the previous one fails?** If so, treat the `GetURL` s as dependent data and use the monadic style. This linear approach will be slower overall than the "applicative" version above, but will also avoid unnecessary I/O.
- **Does the URL for the next site depend on what you download from the previous site?** In this case, you are *forced* to use "monadic" style, because each `GetURL` depends on the output of the previous one.

As you can see, the choice between applicative style and monadic style is not clear cut; it depends on what you want to do.

We'll look at a real implementation of this example in the final post of this series.

but...

It's important to say that just because you *choose* a style doesn't mean it will be implemented as you expect. As we have seen, you can easily implement `apply` in terms of `bind`, so even if you use `<*>` in your code, the implementation may be proceeding monadically.

In the example above, the implementation does not have to run the downloads in parallel. It could run them serially instead. By using applicative style, you're just saying that you don't care about dependencies and so they *could* be downloaded in parallel.

Static vs. dynamic structure

If you use the applicative style, that means that you define all the actions up front -- "statically", as it were.

In the downloading example, the applicative style requires that you specific *in advance* which URLs will be visited. And because there is more knowledge up front it means that we can potentially do things like parallelization or other optimizations.

On the other hand, the monadic style means that only the initial action is known up front. The remainder of the actions are determined dynamically, based on the output of previous actions. This is more flexible, but also limits our ability to see the big picture in advance.

Order of evaluation vs. dependency

Sometimes *dependency* is confused with *order of evaluation*.

Certainly, if one value depends on another then the first value must be evaluated before the second value. And in theory, if the values are completely independent (and have no side effects), then they can be evaluated in any order.

However, even if the values are completely independent, there can still be an *implicit* order in how they are evaluated.

For example, even if the list of `GetURL` s is done in parallel, it's likely that the urls will begin to be fetched in the order in which they are listed, starting with the first one.

And in the `List.apply` implemented in the previous post, we saw that `[f; g] apply [x; y]` resulted in `[f x; f y; g x; g y]` rather than `[f x; g x; f y; g y]` . That is, all the `f` values are first, then all the `g` values.

In general, then, there is a convention that values are evaluated in a left to right order, even if they are independent.

Example: Validation using applicative style and monadic style

To see how both the applicative style and monadic style can be used, let's look at an example using validation.

Say that we have a simple domain containing a `CustomerId` , an `EmailAddress` , and a `CustomerInfo` which is a record containing both of these.

```
type CustomerId = CustomerId of int
type EmailAddress = EmailAddress of string
type CustomerInfo = {
    id: CustomerId
    email: EmailAddress
    }
```

And let's say that there is some validation around creating a `CustomerId`. For example, that the inner `int` must be positive. And of course, there will be some validation around creating a `EmailAddress` too. For example, that it must contain an "@" sign at least.

How would we do this?

First we create a type to represent the success/failure of validation.

```
type Result<'a> =
    | Success of 'a
    | Failure of string list
```

Note that I have defined the `Failure` case to contain a *list* of strings, not just one. This will become important later.

With `Result` in hand, we can go ahead and define the two constructor/validation functions as required:

```
let createCustomerId id =
    if id > 0 then
        Success (CustomerId id)
    else
        Failure ["CustomerId must be positive"]
// int -> Result<CustomerId>

let createEmailAddress str =
    if System.String.IsNullOrEmpty(str) then
        Failure ["Email must not be empty"]
    elif str.Contains("@") then
        Success (EmailAddress str)
    else
        Failure ["Email must contain @-sign"]
// string -> Result<EmailAddress>
```

Notice that `createCustomerId` has type `int -> Result<CustomerId>`, and `createEmailAddress` has type `string -> Result<EmailAddress>`.

That means that both of these validation functions are world-crossing functions, going from the normal world to the `Result<_>` world.

Defining the core functions for `Result`

Since we are dealing with world-crossing functions, we know that we will have to use functions like `apply` and `bind`, so let's define them for our `Result` type.

```
module Result =

    let map f xResult =
        match xResult with
        | Success x ->
            Success (f x)
        | Failure errs ->
            Failure errs
    // Signature: ('a -> 'b) -> Result<'a> -> Result<'b>

    // "return" is a keyword in F#, so abbreviate it
    let retn x =
        Success x
    // Signature: 'a -> Result<'a>

    let apply fResult xResult =
        match fResult,xResult with
        | Success f, Success x ->
            Success (f x)
        | Failure errs, Success x ->
            Failure errs
        | Success f, Failure errs ->
            Failure errs
        | Failure errs1, Failure errs2 ->
            // concat both lists of errors
            Failure (List.concat [errs1; errs2])
    // Signature: Result<('a -> 'b)> -> Result<'a> -> Result<'b>

    let bind f xResult =
        match xResult with
        | Success x ->
            f x
        | Failure errs ->
            Failure errs
    // Signature: ('a -> Result<'b>) -> Result<'a> -> Result<'b>
```

If we check the signatures, we can see that they are exactly as we want:

- `map` has signature: `('a -> 'b) -> Result<'a> -> Result<'b>`
- `retn` has signature: `'a -> Result<'a>`
- `apply` has signature: `Result<('a -> 'b)> -> Result<'a> -> Result<'b>`
- `bind` has signature: `('a -> Result<'b>) -> Result<'a> -> Result<'b>`

I defined a `retn` function in the module to be consistent, but I don't bother to use it very often. The *concept* of `return` is important, but in practice, I'll probably just use the `Success` constructor directly. In languages with type classes, such as Haskell, `return` is used much more.

Also note that `apply` will concat the error messages from each side if both parameters are failures. This allows us to collect all the failures without discarding any. This is the reason why I made the `Failure` case have a list of strings, rather than a single string.

NOTE: I'm using `string` for the failure case to make the demonstration easier. In a more sophisticated design I would list the possible failures explicitly. See my functional error handling talk for more details.

Validation using applicative style

Now that we have have the domain and the toolset around `Result`, let's try using the applicative style to create a `CustomerInfo` record.

The outputs of the validation are already elevated to `Result`, so we know we'll need to use some sort of "lifting" approach to work with them.

First we'll create a function in the normal world that creates a `CustomerInfo` record given a normal `CustomerId` and a normal `EmailAddress`:

```
let createCustomer customerId email =
    { id=customerId;  email=email }
// CustomerId -> EmailAddress -> CustomerInfo
```

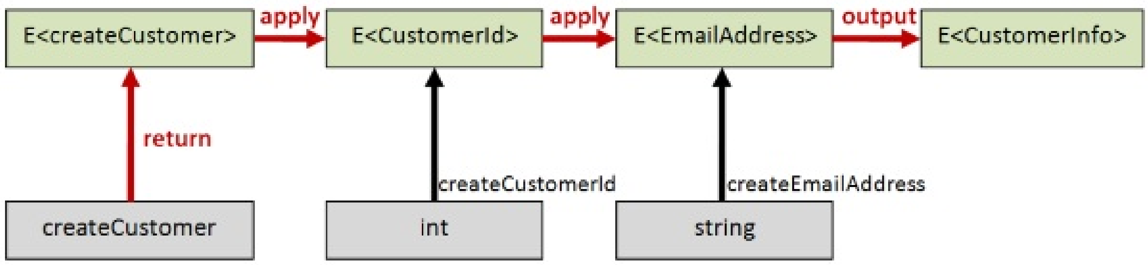
Note that the signature is `CustomerId -> EmailAddress -> CustomerInfo`.

Now we can use the lifting technique with `<!>` and `<*>` that was explained in the previous post:

```
let (<!>) = Result.map
let (<*>) = Result.apply

// applicative version
let createCustomerResultA id email =
    let idResult = createCustomerId id
    let emailResult = createEmailAddress email
    createCustomer <!> idResult <*> emailResult
// int -> string -> Result<CustomerInfo>
```

The signature of this shows that we start with a normal `int` and `string` and return a `Result<CustomerInfo>`



Let's try it out with some good and bad data:

```
let goodId = 1
let badId = 0
let goodEmail = "test@example.com"
let badEmail = "example.com"

let goodCustomerA =
    createCustomerResultA goodId goodEmail
// Result<CustomerInfo> =
//   Success {id = CustomerId 1; email = EmailAddress "test@example.com";}

let badCustomerA =
    createCustomerResultA badId badEmail
// Result<CustomerInfo> =
//   Failure ["CustomerId must be positive"; "Email must contain @-sign"]
```

The `goodCustomerA` is a `Success` and contains the right data, but the `badCustomerA` is a `Failure` and contains two validation error messages. Excellent!

Validation using monadic style

Now let's do another implementation, but this time using monadic style. In this version the logic will be:

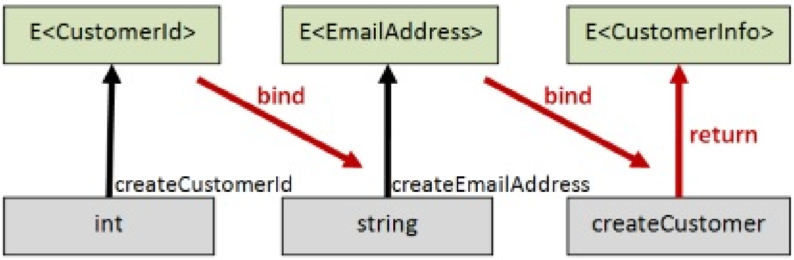
- try to convert an int into a `CustomerId`
- if that is successful, try to convert a string into a `EmailAddress`
- if that is successful, create a `CustomerInfo` from the customerId and email.

Here's the code:

```
let (>>=) x f = Result.bind f x

// monadic version
let createCustomerResultM id email =
    createCustomerId id >>= (fun customerId ->
    createEmailAddress email >>= (fun emailAddress ->
    let customer = createCustomer customerId emailAddress
    Success customer
    ))
// int -> string -> Result<CustomerInfo>
```

The signature of the monadic-style `createCustomerResultM` is exactly the same as the applicative-style `createCustomerResultA` but internally it is doing something different, which will be reflected in the different results we get.




```
let goodCustomerM =
    createCustomerResultM goodId goodEmail
// Result<CustomerInfo> =
//   Success {id = CustomerId 1; email = EmailAddress "test@example.com";}

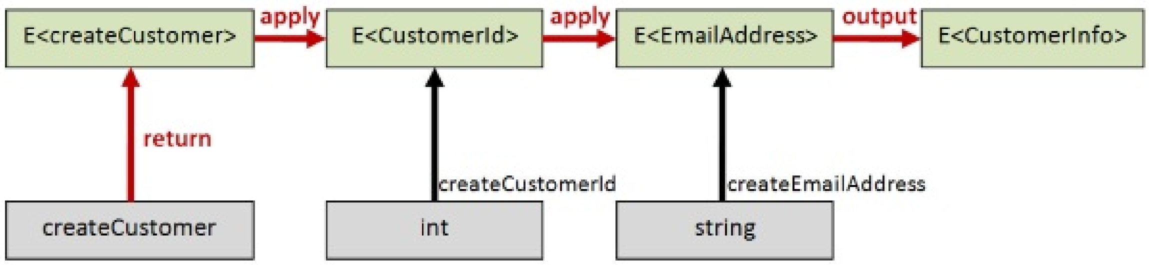
let badCustomerM =
    createCustomerResultM badId badEmail
// Result<CustomerInfo> =
//   Failure ["CustomerId must be positive"]
```

In the good customer case, the end result is the same, but in the bad customer case, only *one* error is returned, the first one. The rest of the validation was short circuited after the `CustomerId` creation failed.

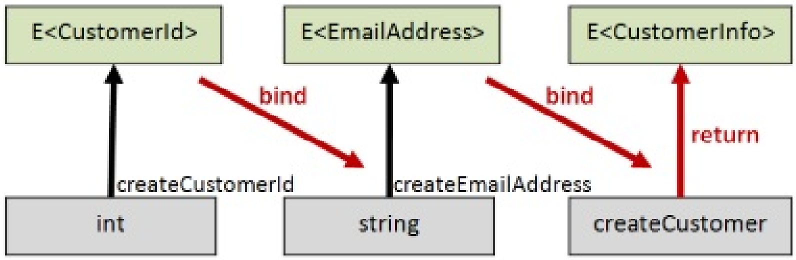
Comparing the two styles

This example has demonstrated the difference between applicative and monadic style quite well, I think.

- The *applicative* example did all the validations up front, and then combined the results. The benefit was that we didn't lose any of the validation errors. The downside was we did work that we might not have needed to do.



- On the other hand, the monadic example did one validation at a time, chained together. The benefit was that we short-circuited the rest of the chain as soon as an error occurred and avoided extra work. The downside was that we only got the *first* error.



Mixing the two styles

Now there is nothing to say that we can't mix and match applicative and monadic styles.

For example, we might build a `CustomerInfo` using applicative style, so that we don't lose any errors, but later on in the program, when a validation is followed by a database update, we probably want to use monadic style, so that the database update is skipped if the validation fails.

Using F# computation expressions

Finally, let's build a computation expression for these `Result` types.

To do this, we just define a class with members called `Return` and `Bind`, and then we create an instance of that class, called `result`, say:

```
module Result =

    type ResultBuilder() =
        member this.Return x = retn x
        member this.Bind(x,f) = bind f x

    let result = new ResultBuilder()
```

We can then rewrite the `createCustomerResultM` function to look like this:

```
let createCustomerResultCE id email = result {
    let! customerId = createCustomerId id
    let! emailAddress = createEmailAddress email
    let customer = createCustomer customerId emailAddress
    return customer }
```

This computation expression version looks almost like using an imperative language.

Note that F# computation expressions are always monadic, as is Haskell do-notation and Scala for-comprehensions. That's not generally a problem, because if you need applicative style it is very easy to write without any language support.

Lifting to a consistent world

In practice, we often have a mish-mash of different kinds of values and functions that we need to combine together.

The trick for doing this is to convert all them to the *same* type, after which they can be combined easily.

Making values consistent

Let's revisit the previous validation example, but let's change the record so that it has an extra property, a `name` of type string:

```
type CustomerId = CustomerId of int
type EmailAddress = EmailAddress of string

type CustomerInfo = {
    id: CustomerId
    name: string  // New!
    email: EmailAddress
    }
```

As before, we want to create a function in the normal world that we will later lift to the `Result` world.

```
let createCustomer customerId name email =
    { id=customerId; name=name; email=email }
// CustomerId -> String -> EmailAddress -> CustomerInfo
```

Now we are ready to update the lifted `createCustomer` with the extra parameter:

```
let (<!>) = Result.map
let (<*>) = Result.apply

let createCustomerResultA id name email =
    let idResult = createCustomerId id
    let emailResult = createEmailAddress email
    createCustomer <!> idResult <*> name <*> emailResult
// ERROR                            ~~~~
```

But this won't compile! In the series of parameters `idResult <*> name <*> emailResult` one of them is not like the others. The problem is that `idResult` and `emailResult` are both Results, but `name` is still a string.

The fix is just to lift `name` into the world of results (say `nameResult`) by using `return` , which for `Result` is just `Success` . Here is the corrected version of the function that does work:

```
let createCustomerResultA id name email =
    let idResult = createCustomerId id
    let emailResult = createEmailAddress email
    let nameResult = Success name  // lift name to Result
    createCustomer <!> idResult <*> nameResult <*> emailResult
```

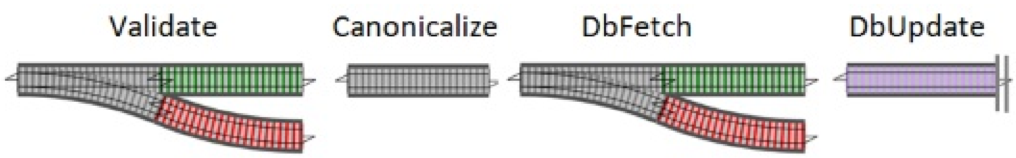
Making functions consistent

The same trick can be used with functions too.

For example, let's say that we have a simple customer update workflow with four steps:

- First, we validate the input. The output of this is the same kind of `Result` type we created above. Note that this validation function could *itself* be the result of combining other, smaller validation functions using `apply` .
- Next, we canonicalize the data. For example: lowercasing emails, trimming whitespace, etc. This step never raises an error.
- Next, we fetch the existing record from the database. For example, getting a customer for the `CustomerId` . This step could fail with an error too.
- Finally, we update the database. This step is a "dead-end" function -- there is no output.

For error handling, I like to think of there being two tracks: a Success track and a Failure track. In this model, an error-generating function is analogous to a railway switch (US) or points (UK).



The problem is that these functions cannot be glued together; they are all different shapes.

The solution is to convert all of them to the *same* shape, in this case the two-track model with success and failure on different tracks. Let's call this *Two-Track world*!

Transforming functions using the toolset

Each original function, then, needs to be elevated to Two-Track world, and we know just the tools that can do this!

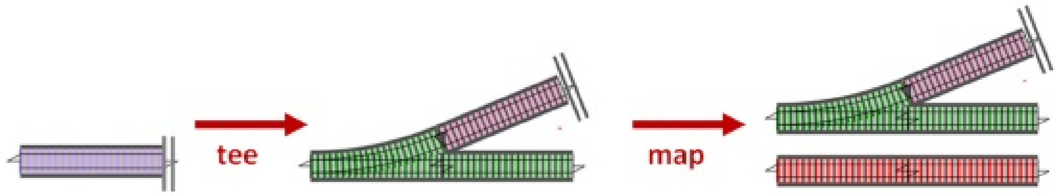
The `Canonicalize` function is a single track function. We can turn it into a two-track function using `map`.



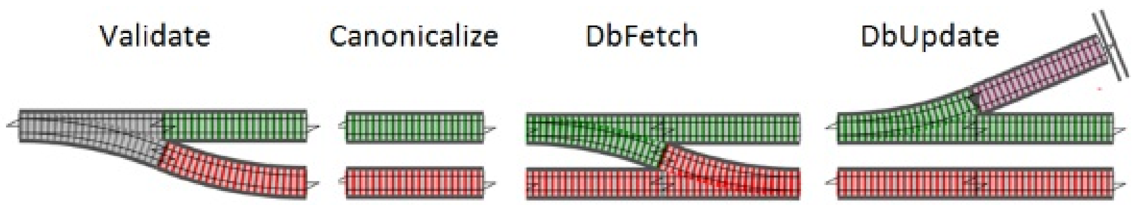
The `DbFetch` function is a world-crossing function. We can turn it into a wholly two-track function using `bind`.



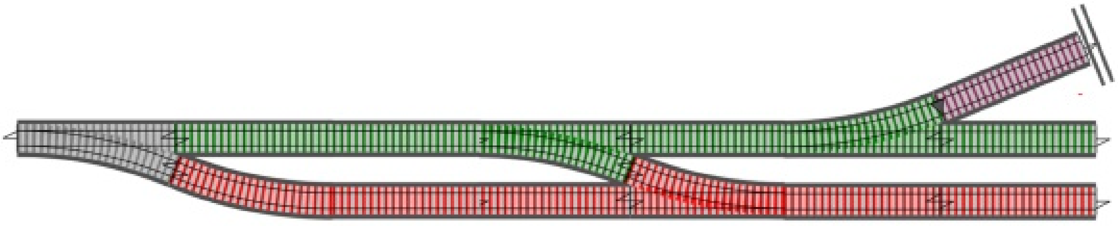
The `DbUpdate` function is more complicated. We don't like dead-end functions, so first we need to transform it to a function where the data keeps flowing. I'll call this function `tee`. The output of `tee` has one track in and one track out, so we need to convert it to a two-track function, again using `map`.



After all these transformations, we can reassemble the new versions of these functions. The result looks like this:



And of course, these functions can now be composed together very easily, so that we end up with a single function looking like this, with one input and a success/failure output:



This combined function is yet another world-crossing function of the form `a->Result<b>` , and so it in turn can be used as a component part of a even bigger function.

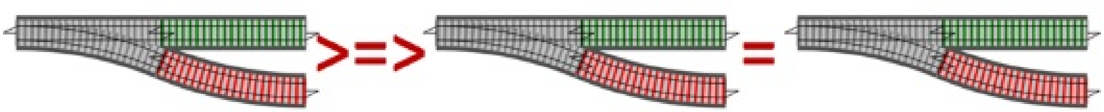
For more examples of this "elevating everything to the same world" approach, see my posts on functional error handling and threading state.

Kleisli world

There is an alternative world which can be used as a basic for consistency which I will call "Kleisli" world, named after Professor Kleisli -- a mathematician, of course!

In Kleisli world *everything* is a cross-world function! Or, using the railway track analogy, everything is a switch (or points).

In Kleisli world, the cross-world functions *can* be composed directly, using an operator called `>=>` for left-to-right composition or `<=<` for right-to-left composition.

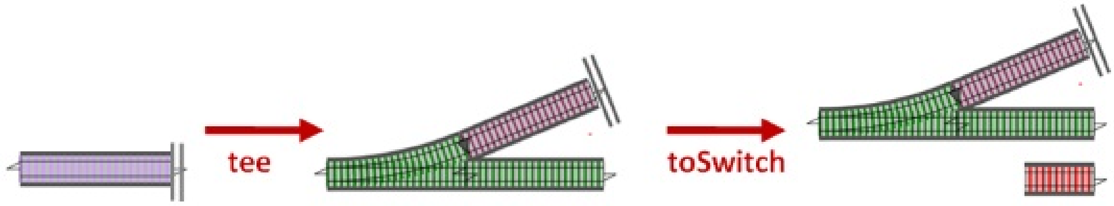


Using the same example as before, we can lift all our functions to Kleisli world.

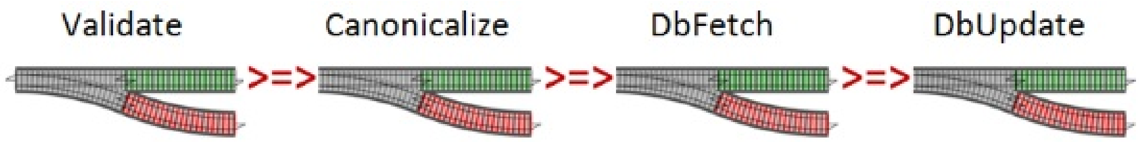
- The `Validate` and `DbFetch` functions are already in the right form so they don't need to be changed.
- The one-track `Canonicalize` function can be lifted to a switch just by lifting the output to a two-track value. Let's call this `toSwitch` .



- The tee-d `DbUpdate` function can be also lifted to a switch just by doing `toSwitch` after the tee.



Once all the functions have been lifted to Kleisli world, they can be composed with Kleisli composition:



Kleisli world has some nice properties that Two-Track world doesn't but on the other hand, I find it hard to get my head around it! So I generally stick to using Two-Track world as my foundation for things like this.

Summary

In this post, we learned about "applicative" vs "monadic" style, and why the choice could have an important effect on which actions are executed, and what results are returned.

We also saw how to lift different kinds values and functions to a a consistent world so that the could be worked with easily.

In the next post we'll look at a common problem: working with lists of elevated values.

Understanding traverse and sequence

This post is one in a series. In the first two posts, I described some of the core functions for dealing with generic data types: `map`, `bind`, and so on. In the previous post, I discussed "applicative" vs "monadic" style, and how to lift values and functions to be consistent with each other.

In this post, we'll look at a common problem: working with lists of elevated values.

Series contents

Here's a list of shortcuts to the various functions mentioned in this series:

- **Part 1: Lifting to the elevated world**
  - The `map` function
  - The `return` function
  - The `apply` function
  - The `liftN` family of functions
  - The `zip` function and ZipList world
- **Part 2: How to compose world-crossing functions**
  - The `bind` function
  - List is not a monad. Option is not a monad.
- **Part 3: Using the core functions in practice**
  - Independent and dependent data
  - Example: Validation using applicative style and monadic style
  - Lifting to a consistent world
  - Kleisli world
- **Part 4: Mixing lists and elevated values**
  - Mixing lists and elevated values
  - The `traverse` / `MapM` function
  - The `sequence` function
  - "Sequence" as a recipe for ad-hoc implementations
  - Readability vs. performance
  - Dude, where's my `filter`?
- **Part 5: A real-world example that uses all the techniques**
  - Example: Downloading and processing a list of websites
  - Treating two worlds as one
- **Part 6: Designing your own elevated world**

  - Designing your own elevated world
  - Filtering out failures
  - The Reader monad
- **Part 7: Summary**
  - List of operators mentioned
  - Further reading

Part 4: Mixing lists and elevated values

A common issue is how to deal with lists or other collections of elevated values.

Here are some examples:

- **Example 1:** We have a `parseInt` with signature `string -> int option`, and we have a list of strings. We want to parse all the strings at once. Now of course we can use `map` to convert the list of strings to a list of options. But what we *really* want is not a "list of options" but an "option of list", a list of parsed ints, wrapped in an option in case any fail.
- **Example 2:** We have a `readCustomerFromDb` function with signature `CustomerId -> Result<Customer>`, that will return `Success` if the record can be found and returned, and `Failure` otherwise. And say we have a list of `CustomerId`s and we want to read all the customers at once. Again, we can use `map` to convert the list of ids to a list of results. But what we *really* want is not a list of `Result<Customer>`, but a `Result` containing a `Customer list`, with the `Failure` case in case of errors.
- **Example 3:** We have a `fetchWebPage` function with signature `Uri -> Async<string>`, that will return a task that will download the page contents on demand. And say we have a list of `Uris`s and we want to fetch all the pages at once. Again, we can use `map` to convert the list of `Uri`s to a list of `Async`s. But what we *really* want is not a list of `Async`, but a `Async` containing a list of strings.

Mapping an Option generating function

Let's start by coming up with a solution for the first case and then seeing if we can generalize it to the others.

The obvious approach would be:

- First, use `map` to turn the list of `string` into a list of `Option<int>`.
- Next, create a function that turns the list of `Option<int>` into an `Option<int list>`.

But this requires *two* passes through the list. Can we do it in one pass?

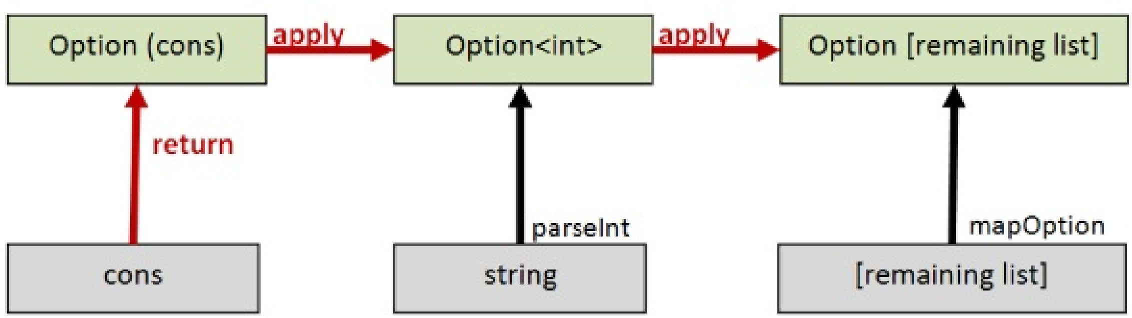
Yes! If we think about how a list is built, there is a "cons" function (`::` in F#) that is used to join the head to the tail. If we elevate this to the `Option` world, we can use `Option.apply` to join a head `Option` to a tail `Option` using the lifted version of `cons` .

```
let (<*>) = Option.apply
let retn = Some

let rec mapOption f list =
    let cons head tail = head :: tail
    match list with
    | [] ->
        retn []
    | head::tail ->
        retn cons <*> (f head) <*> (mapOption f tail)
```

NOTE: I defined `cons` explicitly because `::` is not a function and `List.Cons` takes a tuple and is thus not usable in this context.

Here is the implementation as a diagram:



If you are confused as to how this works, please read the section on `apply` in the first post in this series.

Note also that I am explicitly defining `retn` and using it in the implementation rather than just using `Some` . You'll see why in the next section.

Now let's test it!

```
let parseInt str =
    match (System.Int32.TryParse str) with
    | true,i -> Some i
    | false,_ -> None
// string -> int option

let good = ["1";"2";"3"] |> mapOption parseInt
// Some [1; 2; 3]

let bad = ["1";"x";"y"] |> mapOption parseInt
// None
```

We start by defining `parseInt` of type `string -> int option` (piggybacking on the existing .NET library).

We use `mapOption` to run it against a list of good values, and we get `Some [1; 2; 3]`, with the list *inside* the option, just as we want.

And if we use a list where some of the values are bad, we get `None` for the entire result.

Mapping a Result generating function

Let's repeat this, but this time using the `Result` type from the earlier validation example.

Here's the `mapResult` function:

```
let (<*>) = Result.apply
let retn = Success

let rec mapResult f list =
    let cons head tail = head :: tail
    match list with
    | [] ->
        retn []
    | head::tail ->
        retn cons <*> (f head) <*> (mapResult f tail)
```

Again I am explicitly defining a `retn` rather than just using `Success`. And because of this, the body of the code for `mapResult` and `mapOption` is *exactly the same*!

Now let's change `parseInt` to return a `Result` rather than an `Option`:

```
let parseInt str =
    match (System.Int32.TryParse str) with
    | true,i -> Success i
    | false,_ -> Failure [str + " is not an int"]
```

And then we can rerun the tests again, but this time getting more informative errors in the failure case:

```
let good = ["1";"2";"3"] |> mapResult parseInt
// Success [1; 2; 3]

let bad = ["1";"x";"y"] |> mapResult parseInt
// Failure ["x is not an int"; "y is not an int"]
```

Can we make a generic mapXXX function?

The implementations of `mapOption` and `mapResult` have exactly the same code, the only difference is the different `retn` and `<*>` functions (from Option and Result, respectively).

So the question naturally arises, rather than having `mapResult` , `mapOption` , and other specific implementations for each elevated type, can we make a completely generic version of `mapXXX` that works for *all* elevated types?

The obvious thing would be able to pass these two functions in as an extra parameter, like this:

```
let rec mapE (retn,ap) f list =
    let cons head tail = head :: tail
    let (<*>) = ap

    match list with
    | [] ->
        retn []
    | head::tail ->
        (retn cons) <*> (f head) <*> (mapE retn ap f tail)
```

There are some problems with this though. First, this code doesn't compile in F#! But even if it did, we'd want to make sure that the *same* two parameters were passed around everywhere.

We might attempt this by creating a record structure containing the two parameters, and then create one instance for each type of elevated world:

```
type Applicative<'a,'b> = {
    retn: 'a -> E<'a>
    apply: E<'a->'b> -> E<'a> -> E<'b>
    }

// functions for applying Option
let applOption = {retn = Option.Some; apply=Option.apply}

// functions for applying Result
let applResult = {retn = Result.Success; apply=Result.apply}
```

The instance of the `Applicative` record (`appl` say) would be an extra parameter to our generic `mapE` function, like this:

```
let rec mapE appl f list =
    let cons head tail = head :: tail
    let (<*>) = appl.apply
    let retn = appl.retn

    match list with
    | [] ->
        retn []
    | head::tail ->
        (retn cons) <*> (f head) <*> (mapE retn ap f tail)
```

In use, we would pass in the specific applicative instance that we want, like this:

```
// build an Option specific version...
let mapOption = mapE applOption

// ...and use it
let good = ["1";"2";"3"] |> mapOption parseInt
```

Unfortunately, none of this works either, at least in F#. The `Applicative` type, as defined, won't compile. This is because F# does not support "higher-kinded types". That is, we can't parameterize the `Applicative` type with a generic type, only with concrete types.

In Haskell and languages that *do* support "higher-kinded types", the `Applicative` type that we've defined is similar to a "type class". What's more, with type classes, we don't have to pass around the functions explicitly -- the compiler will do that for us.

There is actually a clever (and hacky) way of getting the same effect in F# though using static type constraints. I'm not going to discuss it here, but you can see it used in the FSharpx library.

The alternative to all this abstraction is just creating a `mapXXX` function for each elevated world that we want to work with: `mapOption` , `mapResult` , `mapAsync` and so on.

Personally I am OK with this cruder approach. There are not that many elevated worlds that you work with regularly, and even though you lose out on abstraction, you gain on explicitness, which is often useful when working in a team of mixed abilities.

So let's look at these `mapXXX` functions, also called `traverse` .

The `traverse` / `mapM` function

Common Names: `mapM` , `traverse` , `for`

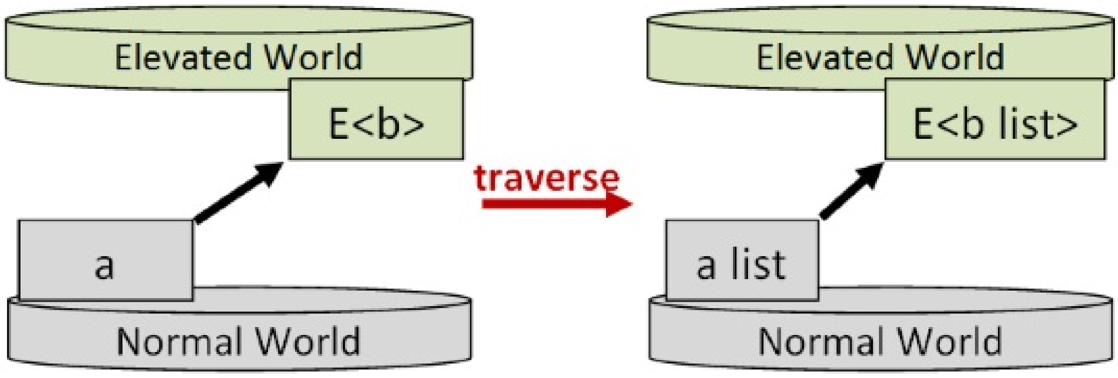
Common Operators: None

What it does: Transforms a world-crossing function into a world-crossing function that works with collections

Signature: `(a->E<b>) -> a list -> E<b list>` (or variants where list is replaced with other collection types)

Description

We saw above that we can define a set of `mapXXX` functions, where XXX stands for an applicative world -- a world that has `apply` and `return`. Each of these `mapXXX` functions transforms a world-crossing function into a world-crossing function that works with collections.



And as we noted above, if the language supports type classes, we can get away with a single implementation, called `mapM` or `traverse`. I'm going to call the general concept `traverse` from now on to make it clear that is different from `map`.

Map vs. Traverse

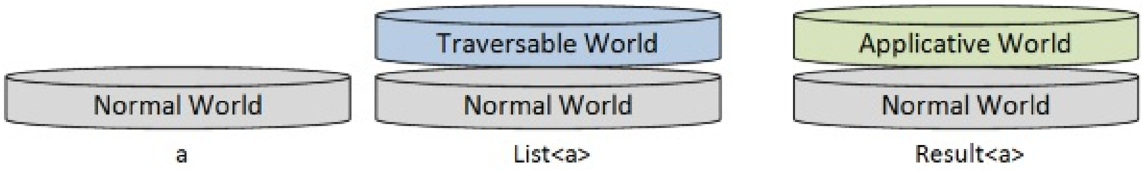
Understanding the difference between `map` and `traverse` can be hard, so let's see if we can explain it in pictures.

First, let's introduce some visual notation using the analogy of an "elevated" world sitting above a "normal" world.

Some of these elevated worlds (in fact almost all of them!) have `apply` and `return` functions. We'll call these "Applicative worlds". Examples include `Option`, `Result`, `Async`, etc.

And some of these elevated worlds have a `traverse` function. We'll call these "Traversable worlds", and we'll use `List` as a classic example.

If a Traversable world is on top, that produces a type such as `List<a>`, and if an Applicative world is on top, that produces a type such as `Result<a>`.



IMPORTANT: I will be using the syntax `List<_>` to represent "List world" for consistency with `Result<_>`, etc. This is not *meant to be the same as the .NET List class! In F#, this would be implemented by the immutable `list` type.*

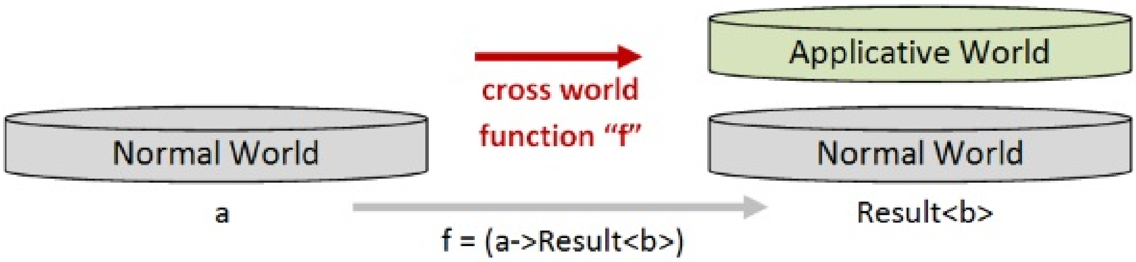
But from now on we are going to be dealing with *both* kinds of elevated worlds in the same "stack".

The Traversable world can be stacked on top of the Applicative world, which produces a type such as `List<Result<a>>`, or alternatively, the Applicative world world can be stacked on top of the Traversable world, which produces a type such as `Result<List<a>>`.

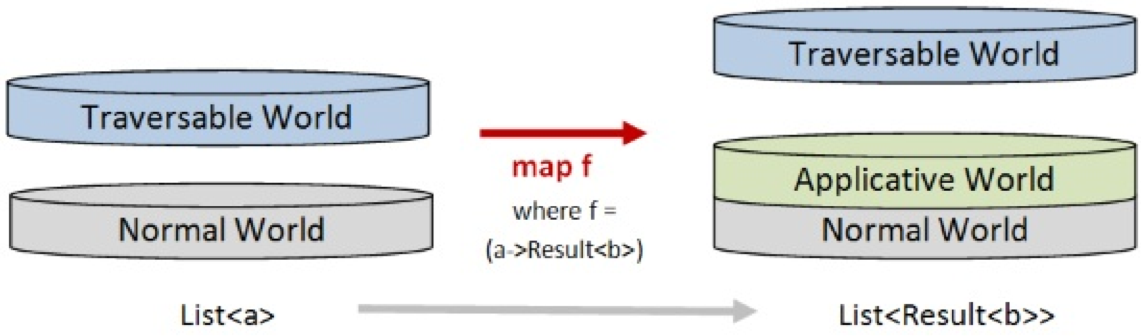


Now let's see what the different kinds of functions look like using this notation.

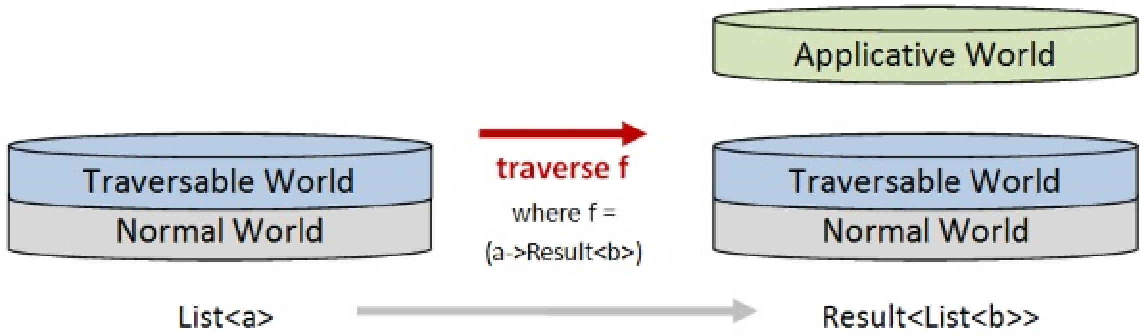
Let's start with a plain cross-world function such as `a -> Result<b>`, where the target world is an applicative world. In the diagram, the input is a normal world (on the left), and the output (on the right) is an applicative world stacked on top of the normal world.



Now if we have a list of normal `a` values, and then we use `map` to transform each `a` value using a function like `a -> Result<b>`, the result will also be a list, but where the contents are `Result<b>` values instead of `a` values.



When it comes to `traverse` the effect is quite different. If we use `traverse` to transform a list of `a` values using that function, the output will be a `Result`, not a list. And the contents of the `Result` will be a `List<b>`.



In other words, with `traverse`, the `List` stays attached to the normal world, and the Applicative world (such as `Result`) is added at the top.

Ok, I know this all sounds very abstract, but it is actually a very useful technique. We'll see an example of this is used in practice below.

Applicative vs. monadic versions of `traverse`

It turns out that `traverse` can be implemented in an applicative style or a monadic style, so there are often two separate implementations to choose from. The applicative versions tend to end in `A` and the monadic versions end in `M`, which is helpful!

Let's see how this works with our trusty `Result` type.

First, we'll implement `traverseResult` using both applicative and monadic approaches.

```
module List =

    /// Map a Result producing function over a list to get a new Result
    /// using applicative style
    /// ('a -> Result<'b>) -> 'a list -> Result<'b list>
    let rec traverseResultA f list =

        // define the applicative functions
        let (<*>) = Result.apply
        let retn = Result.Success

        // define a "cons" function
        let cons head tail = head :: tail

        // loop through the list
        match list with
        | [] ->
            // if empty, lift [] to a Result
            retn []
        | head::tail ->
            // otherwise lift the head to a Result using f
            // and cons it with the lifted version of the remaining list
            retn cons <*> (f head) <*> (traverseResultA f tail)


    /// Map a Result producing function over a list to get a new Result
    /// using monadic style
    /// ('a -> Result<'b>) -> 'a list -> Result<'b list>
    let rec traverseResultM f list =

        // define the monadic functions
        let (>>=) x f = Result.bind f x
        let retn = Result.Success

        // define a "cons" function
        let cons head tail = head :: tail

        // loop through the list
        match list with
        | [] ->
            // if empty, lift [] to a Result
            retn []
        | head::tail ->
            // otherwise lift the head to a Result using f
            // then lift the tail to a Result using traverse
            // then cons the head and tail and return it
            f head                 >>= (fun h ->
            traverseResultM f tail >>= (fun t ->
            retn (cons h t) ))
```

The applicative version is the same implementation that we used earlier.

The monadic version applies the function `f` to the first element and then passes it to `bind`. As always with monadic style, if the result is bad the rest of the list will be skipped.

On the other hand, if the result is good, the next element in the list is processed, and so on. Then the results are cons'ed back together again.

NOTE: These implementations are for demonstration only! Neither of these implementations are tail-recursive, and so they will fail on large lists!

Alright, let's test the two functions and see how they differ. First we need our `parseInt` function:

```
/// parse an int and return a Result
/// string -> Result<int>
let parseInt str =
    match (System.Int32.TryParse str) with
    | true,i -> Result.Success i
    | false,_ -> Result.Failure [str + " is not an int"]
```

Now if we pass in a list of good values (all parsable), the result for both implementations is the same.

```
// pass in strings wrapped in a List
// (applicative version)
let goodA = ["1"; "2"; "3"] |> List.traverseResultA parseInt
// get back a Result containing a list of ints
// Success [1; 2; 3]

// pass in strings wrapped in a List
// (monadic version)
let goodM = ["1"; "2"; "3"] |> List.traverseResultM parseInt
// get back a Result containing a list of ints
// Success [1; 2; 3]
```

But if we pass in a list with some bad values, the results differ.

```
// pass in strings wrapped in a List
// (applicative version)
let badA = ["1"; "x"; "y"] |> List.traverseResultA parseInt
// get back a Result containing a list of ints
// Failure ["x is not an int"; "y is not an int"]

// pass in strings wrapped in a List
// (monadic version)
let badM = ["1"; "x"; "y"] |> List.traverseResultM parseInt
// get back a Result containing a list of ints
// Failure ["x is not an int"]
```

The applicative version returns *all* the errors, while the monadic version returns only the first error.

Implementing `traverse` using `fold`

I mentioned above that the "from-scratch" implementation was not tail recursive and would fail for large lists. That could be fixed of course, at the price of making the code more complicated.

On the other hand, if your collection type has a "right fold" function, as `List` does, then you can use that to make the implementation simpler, faster, and safer too.

In fact, I always like to use `fold` and its ilk wherever possible so that I never have to worry about getting tail-recursion right!

So, here are the re-implementations of `traverseResult`, using `List.foldBack`. I have kept the code as similar as possible, but delegated the looping over the list to the fold function, rather than creating a recursive function.

```
/// Map a Result producing function over a list to get a new Result
/// using applicative style
/// ('a -> Result<'b>) -> 'a list -> Result<'b list>
let traverseResultA f list =

    // define the applicative functions
    let (<*>) = Result.apply
    let retn = Result.Success

    // define a "cons" function
    let cons head tail = head :: tail

    // right fold over the list
    let initState = retn []
    let folder head tail =
        retn cons <*> (f head) <*> tail

    List.foldBack folder list initState

/// Map a Result producing function over a list to get a new Result
/// using monadic style
/// ('a -> Result<'b>) -> 'a list -> Result<'b list>
let traverseResultM f list =

    // define the monadic functions
    let (>>=) x f = Result.bind f x
    let retn = Result.Success

    // define a "cons" function
    let cons head tail = head :: tail

    // right fold over the list
    let initState = retn []
    let folder head tail =
        f head >>= (fun h ->
        tail >>= (fun t ->
        retn (cons h t) ))

    List.foldBack folder list initState
```

Note that this approach will not work for all collection classes. Some types do not have a right fold, so `traverse` must be implemented differently.

What about types other than lists?

All these examples have used the `list` type as the collection type. Can we implement `traverse` for other types too?

Yes. For example, an `Option` can be considered a one-element list, and we can use the same trick.

For example, here's an implementation of `traverseResultA` for `Option`

```
module Option =

    /// Map a Result producing function over an Option to get a new Result
    /// ('a -> Result<'b>) -> 'a option -> Result<'b option>
    let traverseResultA f opt =

        // define the applicative functions
        let (<*>) = Result.apply
        let retn = Result.Success

        // loop through the option
        match opt with
        | None ->
            // if empty, lift None to an Result
            retn None
        | Some x ->
            // lift value to an Result
            (retn Some) <*> (f x)
```

Now we can wrap a string in an `Option` and use `parseInt` on it. Rather than getting a `Option` of `Result`, we invert the stack and get a `Result` of `Option`.

```
// pass in an string wrapped in an Option
let good = Some "1" |> Option.traverseResultA parseInt
// get back a Result containing an Option
// Success (Some 1)
```

If we pass in an unparsable string, we get failure:

```
// pass in an string wrapped in an Option
let bad = Some "x" |> Option.traverseResultA parseInt
// get back a Result containing an Option
// Failure ["x is not an int"]
```

If we pass in `None`, we get `Success` containing `None`!

```
// pass in an string wrapped in an Option
let goodNone = None |> Option.traverseResultA parseInt
// get back a Result containing an Option
// Success (None)
```

This last result might be surprising at first glance, but think of it this way, the parsing didn't fail, so there was no `Failure` at all.

Traversables

Types that can implement a function like `mapXXX` or `traverseXXX` are called *Traversable*. For example, collection types are Traversables as well as some others.

As we saw above, in a language with type classes a Traversable type can get away with just one implementation of `traverse`, but in a language without type classes a Traversable type will need one implementation per elevated type.

Also note that, unlike all the generic functions we have created before, the type being acted on (inside the collection) must have appropriate `apply` and `return` functions in order for `traverse` to be implemented. That is, the inner type must be an Applicative.

The properties of a correct `traverse` implementation

As always, a correct implementation of `traverse` should have some properties that are true no matter what elevated world we are working with.

These are the "Traversable Laws", and a **Traversable** is defined as a generic data type constructor -- `E<T>` -- plus a set of functions (`traverse` or `traverseXXX`) that obey these laws.

The laws are similar to the previous ones. For example, the identity function should be mapped correctly, composition should be preserved, etc.

The `sequence` function

Common Names: `sequence`

Common Operators: None

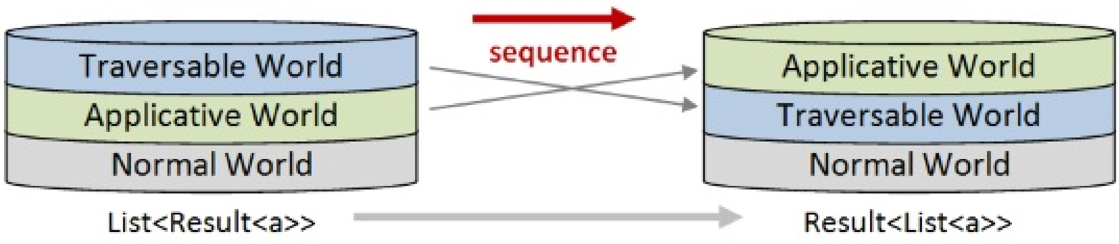
What it does: Transforms a list of elevated values into an elevated value containing a list

Signature: `E<a> list -> E<a list>` (or variants where list is replaced with other collection types)

Description

We saw above how you can use the `traverse` function as a substitute for `map` when you have a function that generates an applicative type such as `Result`.

But what happens if you are just handed a `List<Result>` and you need to change it to a `Result<List>`. That is, you need to swap the order of the worlds on the stack:



This is where `sequence` is useful -- that's exactly what it does! The `sequence` function "swaps layers".

The order of swapping is fixed:

- The Traversable world starts higher and is swapped *down*.
- The Applicative world starts lower and is swapped *up*.

Note that if you aleady have an implementation of `traverse`, then `sequence` can be derived from it easily. In fact, you can think of `sequence` as `traverse` with the `id` function baked in.

Applicative vs Monadic versions of `sequence`

Just as with `traverse`, there can be applicative and monadic versions of `sequence`:

- `sequenceA` for the applicative one.
- `sequenceM` (or just `sequence`) for the monadic one.

A simple example

Let's implement and test a `sequence` implementation for `Result`:

```
module List =

    /// Transform a "list<Result>" into a "Result<list>"
    /// and collect the results using apply
    /// Result<'a> list -> Result<'a list>
    let sequenceResultA x = traverseResultA id x

    /// Transform a "list<Result>" into a "Result<list>"
    /// and collect the results using bind.
    /// Result<'a> list -> Result<'a list>
    let sequenceResultM x = traverseResultM id x
```

Ok, that was too easy! Now let's test it, starting with the applicative version:

```
let goodSequenceA =
    ["1"; "2"; "3"]
    |> List.map parseInt
    |> List.sequenceResultA
// Success [1; 2; 3]

let badSequenceA =
    ["1"; "x"; "y"]
    |> List.map parseInt
    |> List.sequenceResultA
// Failure ["x is not an int"; "y is not an int"]
```

and then the monadic version:

```
let goodSequenceM =
    ["1"; "2"; "3"]
    |> List.map parseInt
    |> List.sequenceResultM
// Success [1; 2; 3]

let badSequenceM =
    ["1"; "x"; "y"]
    |> List.map parseInt
    |> List.sequenceResultM
// Failure ["x is not an int"]
```

As before, we get back a `Result<List>`, and as before the monadic version stops on the first error, while the applicative version accumulates all the errors.

"Sequence" as a recipe for ad-hoc implementations

We saw above that having type classes like Applicative means that you only need to implement `traverse` and `sequence` once. In F# and other languages without high-kinded types you have to create an implementation for each type that you want to traverse over.

Does that mean that the concepts of `traverse` and `sequence` are irrelevant or too abstract? I don't think so.

Instead of thinking of them as library functions, I find that it is useful to think of them as *recipes* -- a set of instructions that you can follow mechanically to solve a particular problem.

In many cases, the problem is unique to a context, and there is no need to create a library function -- you can create a helper function as needed.

Let me demonstrate with an example. Say that you are given a list of options, where each option contains a tuple, like this:

```
let tuples = [Some (1,2); Some (3,4); None; Some (7,8);]
// List<Option<Tuple<int>>>
```

This data is in the form `List<Option<Tuple<int>>>`. And now say, that for some reason, you need to turn it into a *tuple* of two lists, where each list contains options, like this:

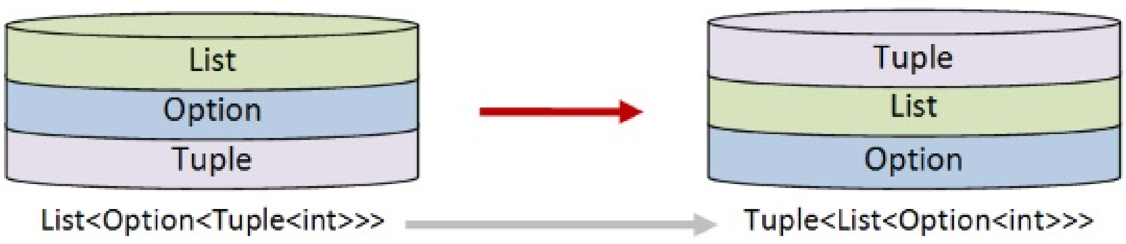
```
let desiredOutput = [Some 1; Some 3; None; Some 7],[Some 2; Some 4; None; Some 8]
// Tuple<List<Option<int>>>
```

The desired result is in the form `Tuple<List<Option<int>>>`.

So, how would you write a function to do this? Quick!

No doubt you could come up with one, but it might require a bit of thought and testing to be sure you get it right.

On the other hand, if you recognize that this task is just transforming a stack of worlds to another stack, you can create a function *mechanically*, almost without thinking.



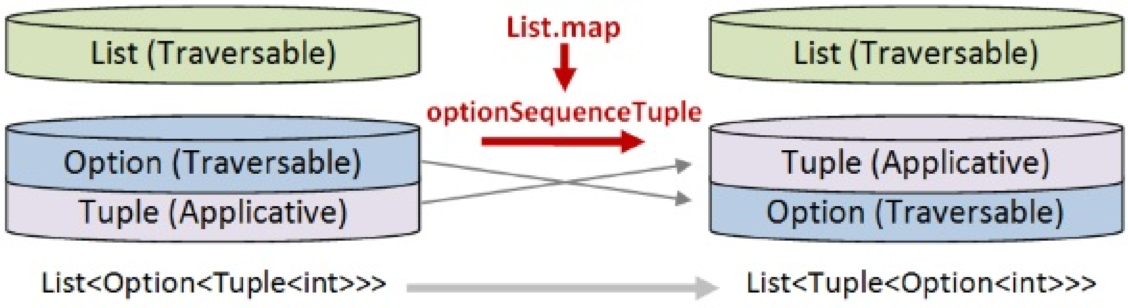
Designing the solution

To design the solution, we need to pay attention to which worlds move up and which worlds move down.

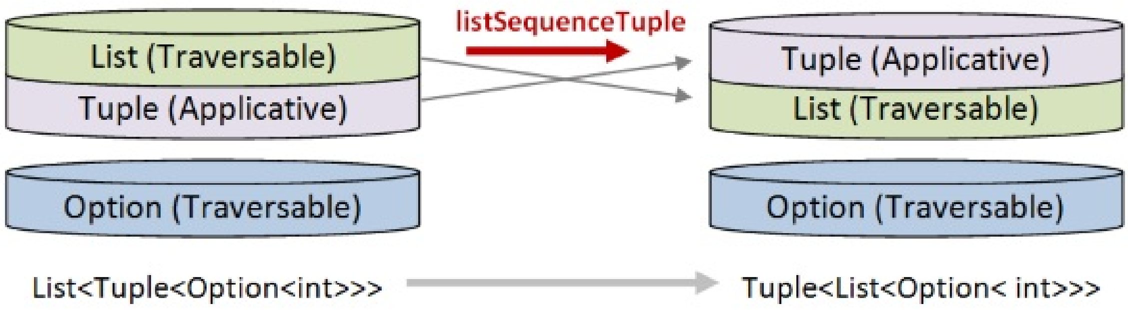
- The tuple world needs to end up at the top, so it will have to be swapped "up", which in turn means that it will play the role of "applicative".
- The option and list worlds need to be swapped "down", which in turn means that they will both play the role of "traversable".

In order to do this transform then, I will need two helper functions:

- `optionSequenceTuple` will move an option down and a tuple up.



- `listSequenceTuple` will move a list down and a tuple up.



Do these helper functions need to be in a library? No. It's unlikely that I will need them again, and even I need them occasionally, I'd prefer to write them scratch to avoid having to take a dependency.

On the other hand, the `List.sequenceResult` function implemented earlier that converts a `List<Result<a>>` to a `Result<List<a>>` is something I do use frequently, and so that one *is* worth centralizing.

Implementing the solution

Once we know how the solution will look, we can start coding mechanically.

First, the tuple is playing the role of the applicative, so we need to define the `apply` and `return` functions:

```
let tupleReturn x = (x, x)
let tupleApply (f,g) (x,y) = (f x, g y)
```

Next, define `listSequenceTuple` using exactly the same right fold template as we did before, with `List` as the traversable and tuple as the applicative:

```
let listSequenceTuple list =
    // define the applicative functions
    let (<*>) = tupleApply
    let retn = tupleReturn

    // define a "cons" function
    let cons head tail = head :: tail

    // right fold over the list
    let initState = retn []
    let folder head tail = retn cons <*> head <*> tail

    List.foldBack folder list initState
```

There is no thinking going on here. I'm just following the template!

We can test it immediately:

```
[ (1,2); (3,4)] |> listSequenceTuple
// Result => ([1; 3], [2; 4])
```

And it gives a tuple with two lists, as expected.

Similarly, define `optionSequenceTuple` using the same right fold template again. This time `Option` is the traversable and tuple is still the applicative:

```
let optionSequenceTuple opt =
    // define the applicative functions
    let (<*>) = tupleApply
    let retn = tupleReturn

    // right fold over the option
    let initState = retn None
    let folder x _ = (retn Some) <*> x

    Option.foldBack folder opt initState
```

We can test it too:

```
Some (1,2) |> optionSequenceTuple
// Result => (Some 1, Some 2)
```

And it gives a tuple with two options, as expected.

Finally, we can glue all the parts together. Again, no thinking required!

```
let convert input =
    input

    // from List<Option<Tuple<int>>> to List<Tuple<Option<int>>>
    |> List.map optionSequenceTuple

    // from List<Tuple<Option<int>>> to Tuple<List<Option<int>>>
    |> listSequenceTuple
```

And if we use it, we get just what we wanted:

```
let output = convert tuples
// ( [Some 1; Some 3; None; Some 7], [Some 2; Some 4; None; Some 8] )

output = desiredOutput |> printfn "Is output correct? %b"
// Is output correct? true
```

Ok, this solution is more work than having one reusable function, but because it is mechanical, it only takes a few minutes to code, and is still easier than trying to come up with your own solution!

Want more? For an example of using `sequence` in a real-world problem, please read this post.

Readability vs. performance

At the beginning of this post I noted our tendency as functional programmers to `map` first and ask questions later.

In other words, given a `Result` producing function like `parseInt`, we would start by collecting the results and only then figure out how to deal with them. Our code would look something like this, then:

```
["1"; "2"; "3"]
|> List.map parseInt
|> List.sequenceResultM
```

But of course, this does involve two passes over the list, and we saw how `traverse` could combine the `map` and the `sequence` in one step, making only one pass over the list, like this:

```
["1"; "2"; "3"]
|> List.traverseResultM parseInt
```

So if `traverse` is more compact and potentially faster, why ever use `sequence`?

Well, sometimes you are given a certain structure and you have no choice, but in other situations I might still prefer the two-step `map-sequence` approach just because it is easier to understand. The mental model for "map" then "swap" seems easier to grasp for most people than the one-step traverse.

In other words, I would always go for readability unless you can prove that performance is impacted. Many people are still learning FP, and being overly cryptic is not helpful, in my experience.

Dude, where's my `filter`?

We've seen that functions like `map` and `sequence` work on lists to transform them into other types, but what about filtering? How can I filter things using these methods?

The answer is -- you can't! `map`, and `traverse` and `sequence` are all "structure preserving". If you start with a list of 10 things you still have a list of 10 things afterwards, albeit somewhere else in the stack. Or if you start with a tree with three branches, you still have a tree of three branches at the end.

In the tuple example above, the original list had four elements, and after the transformation, the two new lists in the tuple also had four elements.

If you need to *change* the structure of a type, you need to use something like `fold`. Fold allows you to build a new structure from an old one, which means that you can use it to create a new list with some elements missing (i.e. a filter).

The various uses of fold are worthy of their own series, so I'm going to save a discussion of filtering for another time.

Summary

In this post, we learned about `traverse` and `sequence` as a way of working with lists of elevated values.

In the next post we'll finish up by working through a practical example that uses all the techniques that have been discussed.

Using map, apply, bind and sequence in practice

This post is the fifth in a series. In the first two posts, I described some of the core functions for dealing with generic data types: `map`, `bind`, and so on. In the third post, I discussed "applicative" vs "monadic" style, and how to lift values and functions to be consistent with each other. In the previous post, I introduced `traverse` and `sequence` as a way of working with lists of elevated values.

In this post, we'll finish up by working through a practical example that uses all the techniques that have been discussed so far.

Series contents

Here's a list of shortcuts to the various functions mentioned in this series:

- **Part 1: Lifting to the elevated world**
  - The `map` function
  - The `return` function
  - The `apply` function
  - The `liftN` family of functions
  - The `zip` function and ZipList world
- **Part 2: How to compose world-crossing functions**
  - The `bind` function
  - List is not a monad. Option is not a monad.
- **Part 3: Using the core functions in practice**
  - Independent and dependent data
  - Example: Validation using applicative style and monadic style
  - Lifting to a consistent world
  - Kleisli world
- **Part 4: Mixing lists and elevated values**
  - Mixing lists and elevated values
  - The `traverse` / `MapM` function
  - The `sequence` function
  - "Sequence" as a recipe for ad-hoc implementations
  - Readability vs. performance
  - Dude, where's my `filter`?
- **Part 5: A real-world example that uses all the techniques**

  - Example: Downloading and processing a list of websites
  - Treating two worlds as one
- **Part 6: Designing your own elevated world**
  - Designing your own elevated world
  - Filtering out failures
  - The Reader monad
- **Part 7: Summary**
  - List of operators mentioned
  - Further reading

Part 5: A real-world example that uses all the techniques

Example: Downloading and processing a list of websites

The example will be a variant of the one mentioned at the beginning of the third post:

- Given a list of websites, create an action that finds the site with the largest home page.

Let's break this down into steps:

First we'll need to transform the urls into a list of actions, where each action downloads the page and gets the size of the content.

And then we need to find the largest content, but in order to do this we'll have to convert the list of actions into a single action containing a list of sizes. And that's where `traverse` or `sequence` will come in.

Let's get started!

The downloader

First we need to create a downloader. I would use the built-in `System.Net.WebClient` class, but for some reason it doesn't allow override of the timeout. I'm going to want to have a small timeout for the later tests on bad uris, so this is important.

One trick is to just subclass `WebClient` and intercept the method that builds a request. So here it is:

```
// define a millisecond Unit of Measure
type [<Measure>] ms

/// Custom implementation of WebClient with settable timeout
type WebClientWithTimeout(timeout:int<ms>) =
    inherit System.Net.WebClient()

    override this.GetWebRequest(address) =
        let result = base.GetWebRequest(address)
        result.Timeout <- int timeout
        result
```

Notice that I'm using units of measure for the timeout value. I find that units of measure are invaluable to distiguish seconds from milliseconds. I once accidentally set a timeout to 2000 seconds rather than 2000 milliseconds and I don't want to make that mistake again!

The next bit of code defines our domain types. We want to be able to keep the url and the size together as we process them. We could use a tuple, but I am a proponent of using types to model your domain, if only for documentation.

```
// The content of a downloaded page
type UriContent =
    UriContent of System.Uri * string

// The content size of a downloaded page
type UriContentSize =
    UriContentSize of System.Uri * int
```

Yes, this might be overkill for a trivial example like this, but in a more serious project I think it is very much worth doing.

Now for the code that does the downloading:

```
/// Get the contents of the page at the given Uri
/// Uri -> Async<Result<UriContent>>
let getUriContent (uri:System.Uri) =
    async {
        use client = new WebClientWithTimeout(1000<ms>) // 1 sec timeout
        try
            printfn "  [%s] Started ..." uri.Host
            let! html = client.AsyncDownloadString(uri)
            printfn "  [%s] ... finished" uri.Host
            let uriContent = UriContent (uri, html)
            return (Result.Success uriContent)
        with
        | ex ->
            printfn "  [%s] ... exception" uri.Host
            let err = sprintf "[%s] %A" uri.Host ex.Message
            return Result.Failure [err ]
        }
```

Notes:

- The .NET library will throw on various errors, so I am catching that and turning it into a `Failure`.
- The `use client =` section ensures that the client will be correctly disposed at the end of the block.
- The whole operation is wrapped in an `async` workflow, and the `let! html = client.AsyncDownloadString` is where the download happens asynchronously.
- I've added some `printfn`s for tracing, just for this example. In real code, I wouldn't do this of course!

Before moving on, let's test this code interactively. First we need a helper to print the result:

```
let showContentResult result =
    match result with
    | Success (UriContent (uri, html)) ->
        printfn "SUCCESS: [%s] First 100 chars: %s" uri.Host (html.Substring(0,100))
    | Failure errs ->
        printfn "FAILURE: %A" errs
```

And then we can try it out on a good site:

```
System.Uri ("http://google.com")
|> getUriContent
|> Async.RunSynchronously
|> showContentResult

//  [google.com] Started ...
//  [google.com] ... finished
// SUCCESS: [google.com] First 100 chars: <!doctype html><html itemscope="" itemtype="
http://schema.org/WebPage" lang="en-GB"><head><meta cont
```

and a bad one:

```
System.Uri ("http://example.bad")
|> getUriContent
|> Async.RunSynchronously
|> showContentResult

//  [example.bad] Started ...
//  [example.bad] ... exception
// FAILURE: ["[example.bad] "The remote name could not be resolved: 'example.bad'""]
```

Extending the Async type with `map` and `apply` and `bind`

At this point, we know that we are going to be dealing with the world of `Async`, so before we go any further, let's make sure that we have our four core functions available:

```
module Async =

    let map f xAsync = async {
        // get the contents of xAsync
        let! x = xAsync
        // apply the function and lift the result
        return f x
        }

    let retn x = async {
        // lift x to an Async
        return x
        }

    let apply fAsync xAsync = async {
        // start the two asyncs in parallel
        let! fChild = Async.StartChild fAsync
        let! xChild = Async.StartChild xAsync

        // wait for the results
        let! f = fChild
        let! x = xChild

        // apply the function to the results
        return f x
        }

    let bind f xAsync = async {
        // get the contents of xAsync
        let! x = xAsync
        // apply the function but don't lift the result
        // as f will return an Async
        return! f x
        }
```

These implementations are straightforward:

- I'm using the `async` workflow to work with `Async` values.
- The `let!` syntax in `map` extracts the content from the `Async` (meaning run it and await the result).
- The `return` syntax in `map`, `retn`, and `apply` lifts the value to an `Async` using `return`.
- The `apply` function runs the two parameters in parallel using a fork/join pattern. If I had instead written `let! fChild = ...` followed by a `let! xChild = ...` that would have been monadic and sequential, which is not what I wanted.
- The `return!` syntax in `bind` means that the value is already lifted and *not* to call `return` on it.

Getting the size of the downloaded page

Getting back on track, we can continue from the downloading step and move on to the process of converting the result to a `UriContentSize` :

```
/// Make a UriContentSize from a UriContent
/// UriContent -> Result<UriContentSize>
let makeContentSize (UriContent (uri, html)) =
    if System.String.IsNullOrEmpty(html) then
        Result.Failure ["empty page"]
    else
        let uriContentSize = UriContentSize (uri, html.Length)
        Result.Success uriContentSize
```

If the input html is null or empty we'll treat this an error, otherwise we'll return a `UriContentSize` .

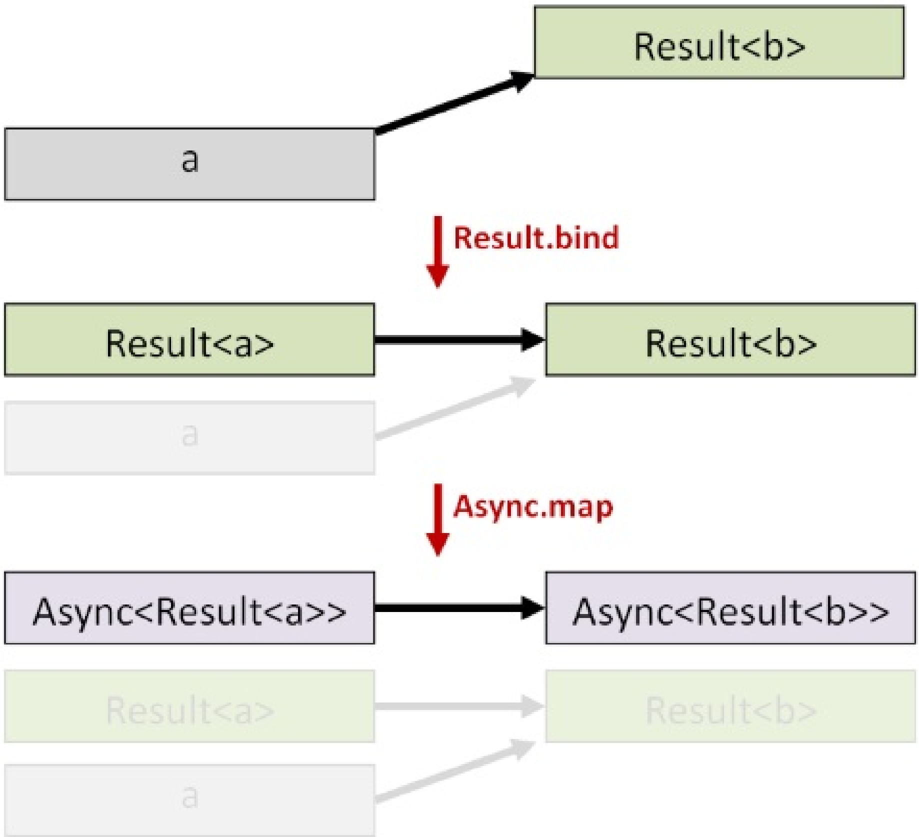
Now we have two functions and we want to combine them into one "get UriContentSize given a Uri" function. The problem is that the outputs and inputs don't match:

- `getUriContent` is `Uri -> Async<Result<UriContent>>`
- `makeContentSize` is `UriContent -> Result<UriContentSize>`

The answer is to transform `makeContentSize` from a function that takes a `UriContent` as input into a function that takes a `Async<Result<UriContent>>` as input. How can we do that?

First, use `Result.bind` to convert it from an `a -> Result<b>` function to a `Result<a> -> Result<b>` function. In this case, `UriContent -> Result<UriContentSize>` becomes `Result<UriContent> -> Result<UriContentSize>` .

Next, use `Async.map` to convert it from an `a -> b` function to a `Async<a> -> Async<b>` function. In this case, `Result<UriContent> -> Result<UriContentSize>` becomes `Async<Result<UriContent>> -> Async<Result<UriContentSize>>` .



And now that it has the right kind of input, so we can compose it with `getUriContent` :

```
/// Get the size of the contents of the page at the given Uri
/// Uri -> Async<Result<UriContentSize>>
let getUriContentSize uri =
    getUriContent uri
    |> Async.map (Result.bind makeContentSize)
```

That's some gnarly type signature, and it's only going to get worse! It's at times like these that I really appreciate type inference.

Let's test again. First a helper to format the result:

```
let showContentSizeResult result =
    match result with
    | Success (UriContentSize (uri, len)) ->
        printfn "SUCCESS: [%s] Content size is %i" uri.Host len
    | Failure errs ->
        printfn "FAILURE: %A" errs
```

And then we can try it out on a good site:

```
System.Uri ("http://google.com")
|> getUriContentSize
|> Async.RunSynchronously
|> showContentSizeResult

//  [google.com] Started ...
//  [google.com] ... finished
//SUCCESS: [google.com] Content size is 44293
```

and a bad one:

```
System.Uri ("http://example.bad")
|> getUriContentSize
|> Async.RunSynchronously
|> showContentSizeResult

//  [example.bad] Started ...
//  [example.bad] ... exception
//FAILURE: ["[example.bad] "The remote name could not be resolved: 'example.bad'""]
```

Getting the largest size from a list

The last step in the process is to find the largest page size.

That's easy. Once we have a list of `UriContentSize` , we can easily find the largest one using `List.maxBy` :

```
/// Get the largest UriContentSize from a list
/// UriContentSize list -> UriContentSize
let maxContentSize list =

    // extract the len field from a UriContentSize
    let contentSize (UriContentSize (_, len)) = len

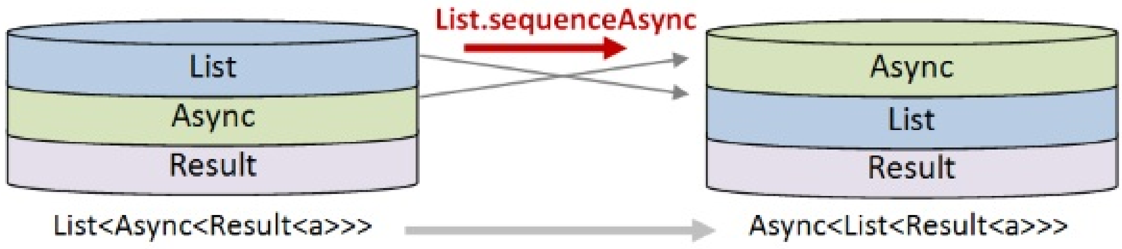
    // use maxBy to find the largest
    list |> List.maxBy contentSize
```

Putting it all together

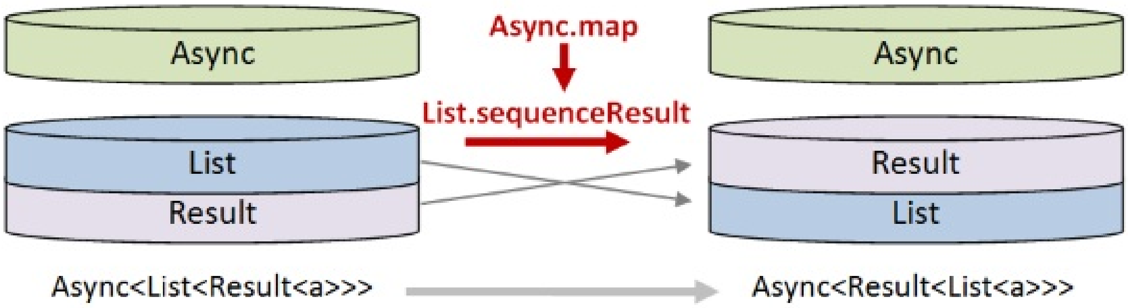
We're ready to assemble all the pieces now, using the following algorithm:

- Start with a list of urls
- Turn the list of strings into a list of uris (`Uri list`)
- Turn the list of `Uri` s into a list of actions (`Async<Result<UriContentSize>> list`)
- Next we need to swap the top two parts of the stack. That is, transform a `List<Async>`

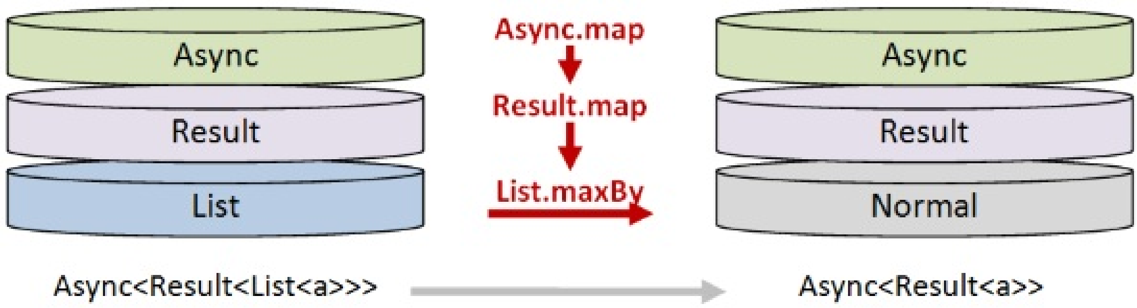
into a `Async<List>` .



- Next we need to swap the *bottom* two parts of the stack -- transform a `List<Result>` into a `Result<List>` . But the two bottom parts of the stack are wrapped in an `Async` so we need to use `Async.map` to do this.



- Finally we need to use `List.maxBy` on the bottom `List` to convert it into a single value. That is, transform a `List<UriContentSize>` into a `UriContentSize` . But the bottom of the stack is wrapped in a `Result` wrapped in an `Async` so we need to use `Async.map` and `Result.map` to do this.



Here's the complete code:

```
/// Get the largest page size from a list of websites
let largestPageSizeA urls =
    urls
    // turn the list of strings into a list of Uris
    // (In F# v4, we can call System.Uri directly!)
    |> List.map (fun s -> System.Uri(s))

    // turn the list of Uris into a "Async<Result<UriContentSize>> list"
    |> List.map getUriContentSize

    // turn the "Async<Result<UriContentSize>> list"
    //   into an "Async<Result<UriContentSize> list>"
    |> List.sequenceAsyncA

    // turn the "Async<Result<UriContentSize> list>"
    //   into a "Async<Result<UriContentSize list>>"
    |> Async.map List.sequenceResultA

    // find the largest in the inner list to get
    //   a "Async<Result<UriContentSize>>"
    |> Async.map (Result.map maxContentSize)
```

This function has signature `string list -> Async<Result<UriContentSize>>`, which is just what we wanted!

There are two `sequence` functions involved here: `sequenceAsyncA` and `sequenceResultA`. The implementations are as you would expect from all the previous discussion, but I'll show the code anyway:

```
module List =

    /// Map a Async producing function over a list to get a new Async
    /// using applicative style
    /// ('a -> Async<'b>) -> 'a list -> Async<'b list>
    let rec traverseAsyncA f list =

        // define the applicative functions
        let (<*>) = Async.apply
        let retn = Async.retn

        // define a "cons" function
        let cons head tail = head :: tail

        // right fold over the list
        let initState = retn []
        let folder head tail =
            retn cons <*> (f head) <*> tail

        List.foldBack folder list initState

    /// Transform a "list<Async>" into a "Async<list>"
    /// and collect the results using apply.
    let sequenceAsyncA x = traverseAsyncA id x

    /// Map a Result producing function over a list to get a new Result
    /// using applicative style
    /// ('a -> Result<'b>) -> 'a list -> Result<'b list>
    let rec traverseResultA f list =

        // define the applicative functions
        let (<*>) = Result.apply
        let retn = Result.Success

        // define a "cons" function
        let cons head tail = head :: tail

        // right fold over the list
        let initState = retn []
        let folder head tail =
            retn cons <*> (f head) <*> tail

        List.foldBack folder list initState

    /// Transform a "list<Result>" into a "Result<list>"
    /// and collect the results using apply.
    let sequenceResultA x = traverseResultA id x
```

Adding a timer

It will be interesting to see how long the download takes for different scenarios, so let's create a little timer that runs a function a certain number of times and takes the average:

```
/// Do countN repetitions of the function f and print the time per run
let time countN label f  =

    let stopwatch = System.Diagnostics.Stopwatch()

    // do a full GC at the start but not thereafter
    // allow garbage to collect for each iteration
    System.GC.Collect()

    printfn "======================="
    printfn "%s" label
    printfn "======================="

    let mutable totalMs = 0L

    for iteration in [1..countN] do
        stopwatch.Restart()
        f()
        stopwatch.Stop()
        printfn "#%2i elapsed:%6ims " iteration stopwatch.ElapsedMilliseconds
        totalMs <- totalMs + stopwatch.ElapsedMilliseconds

    let avgTimePerRun = totalMs / int64 countN
    printfn "%s: Average time per run:%6ims " label avgTimePerRun
```

Ready to download at last

Let's download some sites for real!

We'll define two lists of sites: a "good" one, where all the sites should be accessible, and a "bad" one, containing invalid sites.

```
let goodSites = [
    "http://google.com"
    "http://bbc.co.uk"
    "http://fsharp.org"
    "http://microsoft.com"
    ]

let badSites = [
    "http://example.com/nopage"
    "http://bad.example.com"
    "http://verybad.example.com"
    "http://veryverybad.example.com"
    ]
```

Let's start by running `largestPageSizeA` 10 times with the good sites list:

```
let f() =
    largestPageSizeA goodSites
    |> Async.RunSynchronously
    |> showContentSizeResult
time 10 "largestPageSizeA_Good" f
```

The output is something like this:

```
[google.com] Started ...
[bbc.co.uk] Started ...
[fsharp.org] Started ...
[microsoft.com] Started ...
[bbc.co.uk] ... finished
[fsharp.org] ... finished
[google.com] ... finished
[microsoft.com] ... finished

SUCCESS: [bbc.co.uk] Content size is 108983
largestPageSizeA_Good: Average time per run:   533ms
```

We can see immediately that the downloads are happening in parallel -- they have all started before the first one has finished.

Now what about if some of the sites are bad?

```
let f() =
    largestPageSizeA badSites
    |> Async.RunSynchronously
    |> showContentSizeResult
time 10 "largestPageSizeA_Bad" f
```

The output is something like this:

```
[example.com] Started ...
[bad.example.com] Started ...
[verybad.example.com] Started ...
[veryverybad.example.com] Started ...
[verybad.example.com] ... exception
[veryverybad.example.com] ... exception
[example.com] ... exception
[bad.example.com] ... exception

FAILURE: [
 "[example.com] "The remote server returned an error: (404) Not Found."";
 "[bad.example.com] "The remote name could not be resolved: 'bad.example.com'"";
 "[verybad.example.com] "The remote name could not be resolved: 'verybad.example.com'"
";
 "[veryverybad.example.com] "The remote name could not be resolved: 'veryverybad.examp
le.com'""]

largestPageSizeA_Bad: Average time per run:  2252ms
```

Again, all the downloads are happening in parallel, and all four failures are returned.

Optimizations

The `largestPageSizeA` has a series of maps and sequences in it which means that the list is being iterated over three times and the async mapped over twice.

As I said earlier, I prefer clarity over micro-optimizations unless there is proof otherwise, and so this does not bother me.

However, let's look at what you *could* do if you wanted to.

Here's the original version, with comments removed:

```
let largestPageSizeA urls =
    urls
    |> List.map (fun s -> System.Uri(s))
    |> List.map getUriContentSize
    |> List.sequenceAsyncA
    |> Async.map List.sequenceResultA
    |> Async.map (Result.map maxContentSize)
```

The first two `List.map`s could be combined:

```
let largestPageSizeA urls =
    urls
    |> List.map (fun s -> System.Uri(s) |> getUriContentSize)
    |> List.sequenceAsyncA
    |> Async.map List.sequenceResultA
    |> Async.map (Result.map maxContentSize)
```

The `map-sequence` can be replaced with a `traverse` :

```
let largestPageSizeA urls =
    urls
    |> List.traverseAsyncA (fun s -> System.Uri(s) |> getUriContentSize)
    |> Async.map List.sequenceResultA
    |> Async.map (Result.map maxContentSize)
```

and finally the two `Async.map` s can be combined too:

```
let largestPageSizeA urls =
    urls
    |> List.traverseAsyncA (fun s -> System.Uri(s) |> getUriContentSize)
    |> Async.map (List.sequenceResultA >> Result.map maxContentSize)
```

Personally, I think we've gone too far here. I prefer the original version to this one!

As an aside, one way to get the best of both worlds is to use a "streams" library that automatically merges the maps for you. In F#, a good one is Nessos Streams. Here is a blog post showing the difference between streams and the standard `seq` .

Downloading the monadic way

Let's reimplement the downloading logic using monadic style and see what difference it makes.

First we need a monadic version of the downloader:

```
let largestPageSizeM urls =
    urls
    |> List.map (fun s -> System.Uri(s))
    |> List.map getUriContentSize
    |> List.sequenceAsyncM              // <= "M" version
    |> Async.map List.sequenceResultM   // <= "M" version
    |> Async.map (Result.map maxContentSize)
```

This one uses the monadic `sequence` functions (I won't show them -- the implementation is as you expect).

Let's run `largestPageSizeM` 10 times with the good sites list and see if there is any difference from the applicative version:

```
let f() =
    largestPageSizeM goodSites
    |> Async.RunSynchronously
    |> showContentSizeResult
time 10 "largestPageSizeM_Good" f
```

The output is something like this:

```
  [google.com] Started ...
  [google.com] ... finished
  [bbc.co.uk] Started ...
  [bbc.co.uk] ... finished
  [fsharp.org] Started ...
  [fsharp.org] ... finished
  [microsoft.com] Started ...
  [microsoft.com] ... finished

SUCCESS: [bbc.co.uk] Content size is 108695
largestPageSizeM_Good: Average time per run:   955ms
```

There is a big difference now -- it is obvious that the downloads are happening in series -- each one starts only when the previous one has finished.

As a result, the average time is 955ms per run, almost twice that of the applicative version.

Now what about if some of the sites are bad? What should we expect? Well, because it's monadic, we should expect that after the first error, the remaining sites are skipped, right? Let's see if that happens!

```
let f() =
    largestPageSizeM badSites
    |> Async.RunSynchronously
    |> showContentSizeResult
time 10 "largestPageSizeM_Bad" f
```

The output is something like this:

```
[example.com] Started ...
[example.com] ... exception
[bad.example.com] Started ...
[bad.example.com] ... exception
[verybad.example.com] Started ...
[verybad.example.com] ... exception
[veryverybad.example.com] Started ...
[veryverybad.example.com] ... exception

FAILURE: ["[example.com] "The remote server returned an error: (404) Not Found.""]
largestPageSizeM_Bad: Average time per run:  2371ms
```

Well that was unexpected! All of the sites were visited in series, even though the first one had an error. But in that case, why is only the *first* error returned, rather than *all* the the errors?

Can you see what went wrong?

Explaining the problem

The reason why the implementation did not work as expected is that the chaining of the `Async` s was independent of the chaining of the `Result` s.

If you step through this in a debugger you can see what is happening:

- The first `Async` in the list was run, resulting in a failure.
- `Async.bind` was used with the next `Async` in the list. But `Async.bind` has no concept of error, so the next `Async` was run, producing another failure.
- In this way, all the `Async` s were run, producing a list of failures.
- This list of failures was then traversed using `Result.bind` . Of course, because of the bind, only the first one was processed and the rest ignored.
- The final result was that all the `Async` s were run but only the first failure was returned.

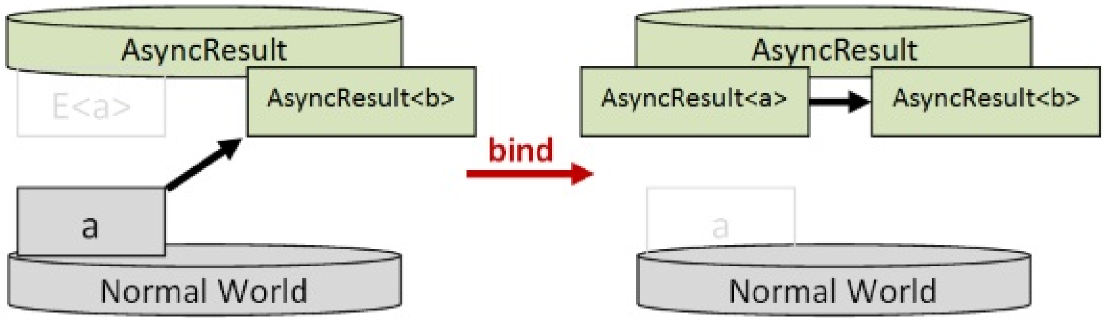
Treating two worlds as one

The fundamental problem is that we are treating the `Async` list and `Result` list as *separate* things to be traversed over. But that means that a failed `Result` has no influence on whether the next `Async` is run.

What we want to do, then, is tie them together so that a bad result *does* determine whether the next `Async` is run.

And in order to do that, we need to treat the `Async` and the `Result` as a *single* type -- let's imaginatively call it `AsyncResult` .

If they are a single type, then `bind` looks like this:



meaning that the previous value will determine the next value.

And also, the "swapping" becomes much simpler:



Defining the AsyncResult type

OK, let's define the `AsyncResult` type and it's associated `map` , `return` , `apply` and `bind` functions.

```
/// type alias (optional)
type AsyncResult<'a> = Async<Result<'a>>

/// functions for AsyncResult
module AsyncResult =
module AsyncResult =

    let map f =
        f |> Result.map |> Async.map

    let retn x =
        x |> Result.retn |> Async.retn

    let apply fAsyncResult xAsyncResult =
        fAsyncResult |> Async.bind (fun fResult ->
        xAsyncResult |> Async.map (fun xResult ->
        Result.apply fResult xResult))

    let bind f xAsyncResult = async {
        let! xResult = xAsyncResult
        match xResult with
        | Success x -> return! f x
        | Failure err -> return (Failure err)
        }
```

Notes:

- The type alias is optional. We can use `Async<Result<'a>>` directly in the code and it wil work fine. The point is that *conceptually* `AsyncResult` is a separate type.
- The `bind` implementation is new. The continuation function `f` is now crossing *two* worlds, and has the signature `'a -> Async<Result<'b>>`.
    - If the inner `Result` is successful, the continuation function `f` is evaluated with the result. The `return!` syntax means that the return value is already lifted.
    - If the inner `Result` is a failure, we have to lift the failure to an Async.

Defining the traverse and sequence functions

With `bind` and `return` in place, we can create the appropriate `traverse` and `sequence` functions for `AsyncResult`:

```
module List =

    /// Map an AsyncResult producing function over a list to get a new AsyncResult
    /// using monadic style
    /// ('a -> AsyncResult<'b>) -> 'a list -> AsyncResult<'b list>
    let rec traverseAsyncResultM f list =

        // define the monadic functions
        let (>>=) x f = AsyncResult.bind f x
        let retn = AsyncResult.retn

        // define a "cons" function
        let cons head tail = head :: tail

        // right fold over the list
        let initState = retn []
        let folder head tail =
            f head >>= (fun h ->
            tail >>= (fun t ->
            retn (cons h t) ))

        List.foldBack folder list initState

    /// Transform a "list<AsyncResult>" into a "AsyncResult<list>"
    /// and collect the results using bind.
    let sequenceAsyncResultM x = traverseAsyncResultM id x
```

Defining and testing the downloading functions

Finally, the `largestPageSize` function is simpler now, with only one sequence needed.

```
let largestPageSizeM_AR urls =
    urls
    |> List.map (fun s -> System.Uri(s) |> getUriContentSize)
    |> List.sequenceAsyncResultM
    |> AsyncResult.map maxContentSize
```

Let's run `largestPageSizeM_AR` 10 times with the good sites list and see if there is any difference from the applicative version:

```
let f() =
    largestPageSizeM_AR goodSites
    |> Async.RunSynchronously
    |> showContentSizeResult
time 10 "largestPageSizeM_AR_Good" f
```

The output is something like this:

```
[google.com] Started ...
[google.com] ... finished
[bbc.co.uk] Started ...
[bbc.co.uk] ... finished
[fsharp.org] Started ...
[fsharp.org] ... finished
[microsoft.com] Started ...
[microsoft.com] ... finished

SUCCESS: [bbc.co.uk] Content size is 108510
largestPageSizeM_AR_Good: Average time per run:  1026ms
```

Again, the downloads are happening in series. And again, the time per run is almost twice that of the applicative version.

And now the moment we've been waiting for! Will it skip the downloading after the first bad site?

```
let f() =
    largestPageSizeM_AR badSites
    |> Async.RunSynchronously
    |> showContentSizeResult
time 10 "largestPageSizeM_AR_Bad" f
```

The output is something like this:

```
  [example.com] Started ...
  [example.com] ... exception

FAILURE: ["[example.com] "The remote server returned an error: (404) Not Found.""]
largestPageSizeM_AR_Bad: Average time per run:   117ms
```

Success! The error from the first bad site prevented the rest of the downloads, and the short run time is proof of that.

Summary

In this post, we worked through a small practical example. I hope that this example demonstrated that `map`, `apply`, `bind`, `traverse`, and `sequence` are not just academic abstractions but essential tools in your toolbelt.

In the next post we'll working through another practical example, but this time we will end up creating our *own* elevated world. See you then!

Reinventing the Reader monad

This post is the sixth in a series. In the first two posts, I described some of the core functions for dealing with generic data types: `map`, `bind`, and so on. In the third post, I discussed "applicative" vs "monadic" style, and how to lift values and functions to be consistent with each other. In the fourth and previous posts, I introduced `traverse` and `sequence` as a way of working with lists of elevated values, and we saw this used in a practical example: downloading some URLs.

In this post, we'll finish up by working through another practical example, but this time we'll create our own "elevated world" as a way to deal with awkward code. We'll see that this approach is so common that it has a name -- the "Reader monad".

Series contents

Here's a list of shortcuts to the various functions mentioned in this series:

- **Part 1: Lifting to the elevated world**
  - The `map` function
  - The `return` function
  - The `apply` function
  - The `liftN` family of functions
  - The `zip` function and ZipList world
- **Part 2: How to compose world-crossing functions**
  - The `bind` function
  - List is not a monad. Option is not a monad.
- **Part 3: Using the core functions in practice**
  - Independent and dependent data
  - Example: Validation using applicative style and monadic style
  - Lifting to a consistent world
  - Kleisli world
- **Part 4: Mixing lists and elevated values**
  - Mixing lists and elevated values
  - The `traverse` / `MapM` function
  - The `sequence` function
  - "Sequence" as a recipe for ad-hoc implementations
  - Readability vs. performance
  - Dude, where's my `filter`?

- **Part 5: A real-world example that uses all the techniques**
  - Example: Downloading and processing a list of websites
  - Treating two worlds as one
- **Part 6: Designing your own elevated world**
  - Designing your own elevated world
  - Filtering out failures
  - The Reader monad
- **Part 7: Summary**
  - List of operators mentioned
  - Further reading

Part 6: Designing your own elevated world

The scenario we'll be working with in this post is just this:

A customer comes to your site and wants to view information about the products they have purchased.

In this example, we'll assume that you have a API for a key/value store (such as Redis or a NoSql database), and all the information you need is stored there.

So the code we need will look something like this:

```
Open API connection
Get product ids purchased by customer id using the API
For each product id:
    get the product info for that id using the API
Close API connection
Return the list of product infos
```

How hard can that be?

Well, it turns out to be surprisingly tricky! Luckily, we can find a way to make it easier using the concepts in this series.

Defining the domain and a dummy ApiClient

First let's define the domain types:

- There will be a `CustomerId` and `ProductId` of course.

- For the product information, we'll just define a simple `ProductInfo` with a `ProductName` field.

Here are the types:

```
type CustId = CustId of string
type ProductId = ProductId of string
type ProductInfo = {ProductName: string; }
```

For testing our api, let's create an `ApiClient` class with some `Get` and `Set` methods, backed by a static mutable dictionary. This is based on similar APIs such as the Redis client.

Notes:

- The `Get` and `Set` both work with objects, so I've added a casting mechanism.
- In case of errors such as a failed cast, or a missing key, I'm using the `Result` type that we've been using throughout this series. Therefore, both `Get` and `Set` return `Result` s rather than plain objects.
- To make it more realistic, I've also added dummy methods for `Open` , `Close` and `Dispose` .
- All methods trace a log to the console.

```
type ApiClient() =
    // static storage
    static let mutable data = Map.empty<string,obj>

    /// Try casting a value
    /// Return Success of the value or Failure on failure
    member private this.TryCast<'a> key (value:obj) =
        match value with
        | :? 'a as a ->
            Result.Success a
        | _  ->
            let typeName = typeof<'a>.Name
            Result.Failure [sprintf "Can't cast value at %s to %s" key typeName]

    /// Get a value
    member this.Get<'a> (id:obj) =
        let key =  sprintf "%A" id
        printfn "[API] Get %s" key
        match Map.tryFind key data with
        | Some o ->
            this.TryCast<'a> key o
        | None ->
            Result.Failure [sprintf "Key %s not found" key]

    /// Set a value
    member this.Set (id:obj) (value:obj) =
        let key =  sprintf "%A" id
        printfn "[API] Set %s" key
        if key = "bad" then  // for testing failure paths
            Result.Failure [sprintf "Bad Key %s " key]
        else
            data <- Map.add key value data
            Result.Success ()

    member this.Open() =
        printfn "[API] Opening"

    member this.Close() =
        printfn "[API] Closing"

    interface System.IDisposable with
        member this.Dispose() =
            printfn "[API] Disposing"
```

Let's do some tests:

```
do
    use api = new ApiClient()
    api.Get "K1" |> printfn "[K1] %A"

    api.Set "K2" "hello" |> ignore
    api.Get<string> "K2" |> printfn "[K2] %A"

    api.Set "K3" "hello" |> ignore
    api.Get<int> "K3" |> printfn "[K3] %A"
```

And the results are:

```
[API] Get "K1"
[K1] Failure ["Key "K1" not found"]
[API] Set "K2"
[API] Get "K2"
[K2] Success "hello"
[API] Set "K3"
[API] Get "K3"
[K3] Failure ["Can't cast value at "K3" to Int32"]
[API] Disposing
```

A first implementation attempt

For our first attempt at implementing the scenario, let's start with the pseudo-code from above:

```
let getPurchaseInfo (custId:CustId) : Result<ProductInfo list> =

    // Open api connection
    use api = new ApiClient()
    api.Open()

    // Get product ids purchased by customer id
    let productIdsResult = api.Get<ProductId list> custId

    let productInfosResult = ??

    // Close api connection
    api.Close()

    // Return the list of product infos
    productInfosResult
```

So far so good, but there is a bit of a problem already.

The `getPurchaseInfo` function takes a `CustId` as input, but it can't just output a list of `ProductInfo` s, because there might be a failure. That means that the return type needs to be `Result<ProductInfo list>` .

Ok, how do we create our `productInfosResult` ?

Well that should be easy. If the `productIdsResult` is Success, then loop through each id and get the info for each id. If the `productIdsResult` is Failure, then just return that failure.

```
let getPurchaseInfo (custId:CustId) : Result<ProductInfo list> =

    // Open api connection
    use api = new ApiClient()
    api.Open()

    // Get product ids purchased by customer id
    let productIdsResult = api.Get<ProductId list> custId

    let productInfosResult =
        match productIdsResult with
        | Success productIds ->
            let productInfos = ResizeArray()  // Same as .NET List<T>
            for productId in productIds do
                let productInfo = api.Get<ProductInfo> productId
                productInfos.Add productInfo  // mutation!
            Success productInfos
        | Failure err ->
            Failure err

    // Close api connection
    api.Close()

    // Return the list of product infos
    productInfosResult
```

Hmmm. It's looking a bit ugly. And I'm having to use a mutable data structure (`productInfos`) to accumulate each product info and then wrap it in `Success` .

And there's a worse problem The `productInfo` that I'm getting from `api.Get<ProductInfo>` is not a `ProductInfo` at all, but a `Result<ProductInfo>` , so `productInfos` is not the right type at all!

Let's add code to test each `ProductInfo` result. If it's a success, then add it to the list of product infos, and if it's a failure, then return the failure.

```
let getPurchaseInfo (custId:CustId) : Result<ProductInfo list> =

    // Open api connection
    use api = new ApiClient()
    api.Open()

    // Get product ids purchased by customer id
    let productIdsResult = api.Get<ProductId list> custId

    let productInfosResult =
        match productIdsResult with
        | Success productIds ->
            let productInfos = ResizeArray()  // Same as .NET List<T>
            let mutable anyFailures = false
            for productId in productIds do
                let productInfoResult = api.Get<ProductInfo> productId
                match productInfoResult with
                | Success productInfo ->
                    productInfos.Add productInfo
                | Failure err ->
                    Failure err
            Success productInfos
        | Failure err ->
            Failure err

    // Close api connection
    api.Close()

    // Return the list of product infos
    productInfosResult
```

Um, no. That won't work at all. The code above will not compile. We can't do an "early return" in the loop when a failure happens.

So what do we have so far? Some really ugly code that won't even compile.

There has to be a better way.

A second implementation attempt

It would be great if we could hide all this unwrapping and testing of `Result` s. And there is -- computation expressions to the rescue.

If we create a computation expression for `Result` we can write the code like this:

```
/// CustId -> Result<ProductInfo list>
let getPurchaseInfo (custId:CustId) : Result<ProductInfo list> =

    // Open api connection
    use api = new ApiClient()
    api.Open()

    let productInfosResult = Result.result {

        // Get product ids purchased by customer id
        let! productIds = api.Get<ProductId list> custId

        let productInfos = ResizeArray()  // Same as .NET List<T>
        for productId in productIds do
            let! productInfo = api.Get<ProductInfo> productId
            productInfos.Add productInfo
        return productInfos |> List.ofSeq
        }

    // Close api connection
    api.Close()

    // Return the list of product infos
    productInfosResult
```

In `let productInfosResult = Result.result { .. }` code we create a `result` computation expression that simplifies all the unwrapping (with `let!`) and wrapping (with `return`).

And so this implementation has no explicit `xxxResult` values anywhere. However, it still has to use a mutable collection class to do the accumulation, because the `for productId in productIds do` is not actually a real `for` loop, and we can't replace it with `List.map`, say.

The `result` computation expression.

Which brings us onto the implementation of the `result` computation expression. In the previous posts, `ResultBuilder` only had two methods, `Return` and `Bind`, but in order to get the `for..in..do` functionality, we have to implement a lot of other methods too, and it ends up being a bit more complicated.

```
module Result =

    let bind f xResult = ...

    type ResultBuilder() =
        member this.Return x = retn x
        member this.ReturnFrom(m: Result<'T>) = m
        member this.Bind(x,f) = bind f x

        member this.Zero() = Failure []
        member this.Combine (x,f) = bind f x
        member this.Delay(f: unit -> _) = f
        member this.Run(f) = f()

        member this.TryFinally(m, compensation) =
            try this.ReturnFrom(m)
            finally compensation()

        member this.Using(res:#System.IDisposable, body) =
            this.TryFinally(body res, fun () ->
            match res with
            | null -> ()
            | disp -> disp.Dispose())

        member this.While(guard, f) =
            if not (guard()) then
                this.Zero()
            else
                this.Bind(f(), fun _ -> this.While(guard, f))

        member this.For(sequence:seq<_>, body) =
            this.Using(sequence.GetEnumerator(), fun enum ->
                this.While(enum.MoveNext, this.Delay(fun () ->
                    body enum.Current)))

    let result = new ResultBuilder()
```

I have a series about the internals of computation expressions, so I don't want to explain all that code here. Instead, for the rest of the post we'll work on refactoring `getPurchaseInfo`, and by the end of it we'll see that we don't need the `result` computation expression at all.

Refactoring the function

The problem with the `getPurchaseInfo` function as it stands is that it mixes concerns: it both creates the `ApiClient` and does some work with it.

There a number of problems with this approach:

- If we want to do different work with the API, we have to repeat the open/close part of this code. And it's possible that one of the implementations might open the API but forget to close it.
- It's not testable with a mock API client.

We can solve both of these problems by separating the creation of an `ApiClient` from its use by parameterizing the action, like this.

```
let executeApiAction apiAction  =

    // Open api connection
    use api = new ApiClient()
    api.Open()

    // do something with it
    let result = apiAction api

    // Close api connection
    api.Close()

    // return result
    result
```

The action function that is passed in would look like this, with a parameter for the `ApiClient` as well as for the `CustId` :

```
/// CustId -> ApiClient -> Result<ProductInfo list>
let getPurchaseInfo (custId:CustId) (api:ApiClient) =

    let productInfosResult = Result.result {
        let! productIds = api.Get<ProductId list> custId

        let productInfos = ResizeArray()  // Same as .NET List<T>
        for productId in productIds do
            let! productInfo = api.Get<ProductInfo> productId
            productInfos.Add productInfo
        return productInfos |> List.ofSeq
        }

    // return result
    productInfosResult
```

Note that `getPurchaseInfo` has *two* parameters, but `executeApiAction` expects a function with only one.

No problem! Just use partial application to bake in the first parameter:

```
let action = getPurchaseInfo (CustId "C1")  // partially apply
executeApiAction action
```

That's why the `ApiClient` is the *second* parameter in the parameter list -- so that we can do partial application.

More refactoring

We might need to get the product ids for some other purpose, and also the productInfo, so let's refactor those out into separate functions too:

```
/// CustId -> ApiClient -> Result<ProductId list>
let getPurchaseIds (custId:CustId) (api:ApiClient) =
    api.Get<ProductId list> custId

/// ProductId -> ApiClient -> Result<ProductInfo>
let getProductInfo (productId:ProductId) (api:ApiClient) =
    api.Get<ProductInfo> productId

/// CustId -> ApiClient -> Result<ProductInfo list>
let getPurchaseInfo (custId:CustId) (api:ApiClient) =

    let result = Result.result {
        let! productIds = getPurchaseIds custId api

        let productInfos = ResizeArray()
        for productId in productIds do
            let! productInfo = getProductInfo productId api
            productInfos.Add productInfo
        return productInfos |> List.ofSeq
        }

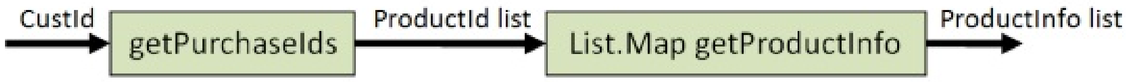
    // return result
    result
```

Now, we have these nice core functions `getPurchaseIds` and `getProductInfo`, but I'm annoyed that I have to write messy code to glue them together in `getPurchaseInfo`.

Ideally, what I'd like to do is pipe the output of `getPurchaseIds` into `getProductInfo` like this:

```
let getPurchaseInfo (custId:CustId) =
    custId
    |> getPurchaseIds
    |> List.map getProductInfo
```

Or as a diagram:



But I can't, and there are two reasons why:

- First, `getProductInfo` has *two* parameters. Not just a `ProductId` but also the `ApiClient`.
- Second, even if `ApiClient` wasn't there, the input of `getProductInfo` is a simple `ProductId` but the output of `getPurchaseIds` is a `Result`.

Wouldn't it be great if we could solve both of these problems!

Introducing our own elevated world

Let's address the first problem. How can we compose functions when the extra `ApiClient` parameter keeps getting in the way?

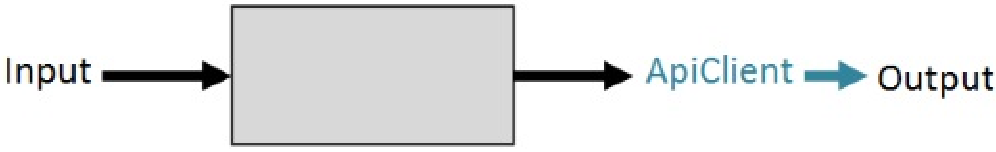
This is what a typical API calling function looks like:



If we look at the type signature we see this, a function with two parameters:

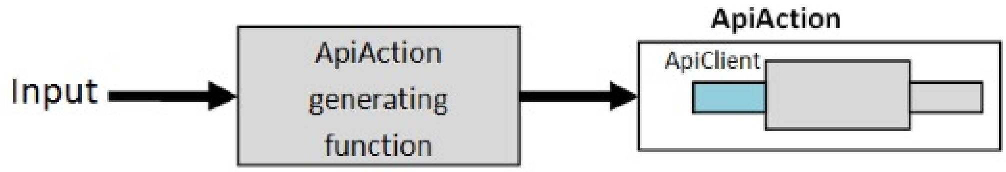


But *another* way to interpret this function is as a function with *one* parameter that returns another function. The returned function has an `ApiClient` parameter and returns the final ouput.



You might think of it like this: I have an input right now, but I won't have an actual `ApiClient` until later, so let me use the input to create a api-consuming function that can I glue together in various ways right now, without needing a `ApiClient` at all.

Let's give this api-consuming function a name. Let's call it `ApiAction`.



In fact, let's do more than that -- let's make it a type!

```
type ApiAction<'a> = (ApiClient -> 'a)
```

Unfortunately, as it stands, this is just a type alias for a function, not a separate type. We need to wrap it in a single case union to make it a distinct type.

```
type ApiAction<'a> = ApiAction of (ApiClient -> 'a)
```

Rewriting to use ApiAction

Now that we have a real type to use, we can rewrite our core domain functions to use it.

First `getPurchaseIds` :

```
// CustId -> ApiAction<Result<ProductId list>>
let getPurchaseIds (custId:CustId) =

    // create the api-consuming function
    let action (api:ApiClient) =
        api.Get<ProductId list> custId

    // wrap it in the single case
    ApiAction action
```

The signature is now `CustId -> ApiAction<Result<ProductId list>>` , which you can interpret as meaning: "give me a CustId and I will give a you a ApiAction that, when given an api, will make a list of ProductIds".

Similarly, `getProductInfo` can be rewritten to return an `ApiAction` :

```
// ProductId -> ApiAction<Result<ProductInfo>>
let getProductInfo (productId:ProductId) =

    // create the api-consuming function
    let action (api:ApiClient) =
        api.Get<ProductInfo> productId

    // wrap it in the single case
    ApiAction action
```

Notice those signatures:

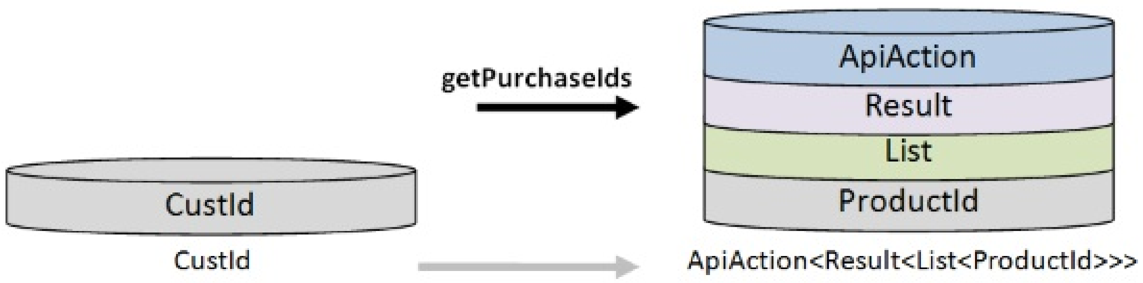
- `CustId -> ApiAction<Result<ProductId list>>`
- `ProductId -> ApiAction<Result<ProductInfo>>`

This is starting to look awfully familiar. Didn't we see something just like this in the previous post, with `Async<Result<_>>` ?

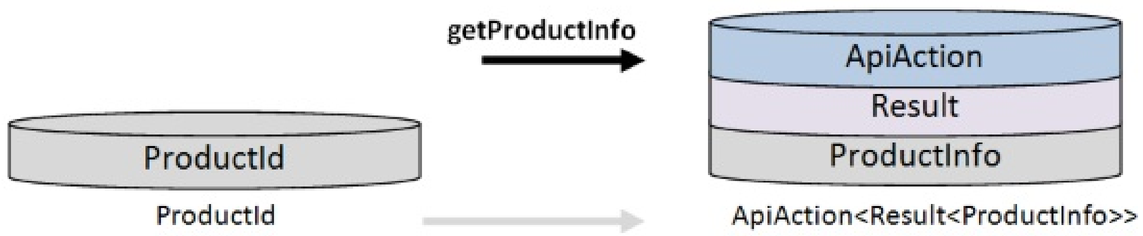
ApiAction as an elevated world

If we draw diagrams of the various types involved in these two functions, we can clearly see that `ApiAction` is an elevated world, just like `List` and `Result` . And that means that we should be able to use the *same* techniques as we have used before: `map` , `bind` , `traverse` , etc.

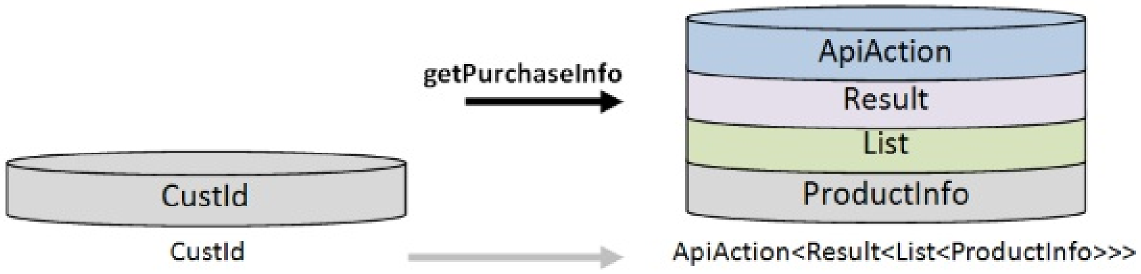
Here's `getPurchaseIds` as a stack diagram. The input is a `CustId` and the output is an `ApiAction<Result<List<ProductId>>>` :



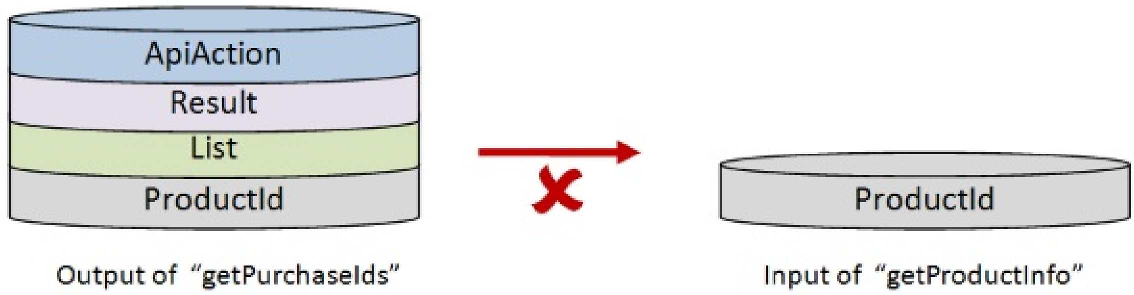
and with `getProductInfo` the input is a `ProductId` and the output is an `ApiAction<Result<ProductInfo>>` :



The combined function that we want, `getPurchaseInfo` , should look like this:



And now the problem in composing the two functions is very clear: the output of `getPurchaseIds` can not be used as the input for `getProductInfo` :



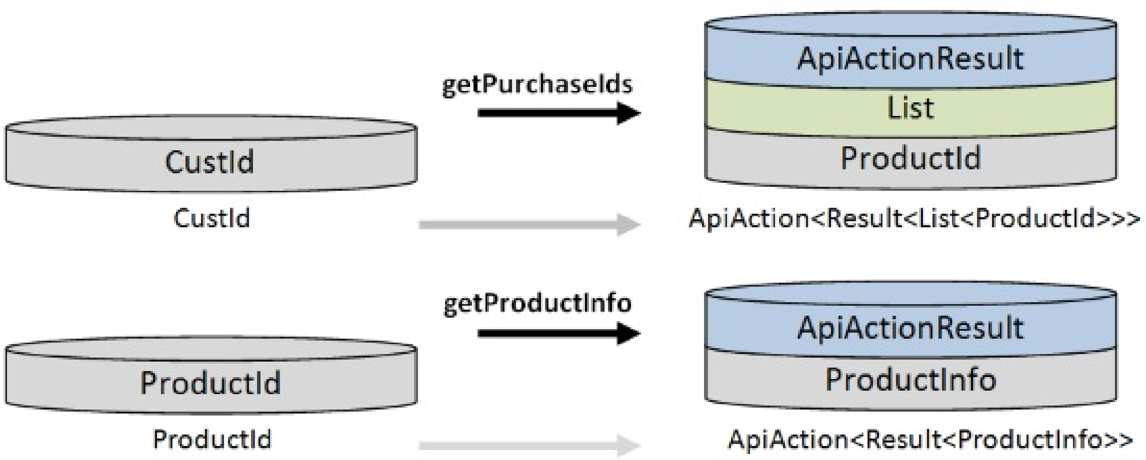
But I think that you can see that we have some hope! There should be some way of manipulating these layers so that they *do* match up, and then we can compose them easily.

So that's what we will work on next.

Introducting ApiActionResult

In the last post we merged `Async` and `Result` into the compound type `AsyncResult` . We can do the same here, and create the type `ApiActionResult` .

When we make this change, our two functions become slightly simpler:



Enough diagrams -- let's write some code now.

First, we need to define `map` , `apply` , `return` and `bind` for `ApiAction` :

```
module ApiAction =

    /// Evaluate the action with a given api
    /// ApiClient -> ApiAction<'a> -> 'a
    let run api (ApiAction action) =
        let resultOfAction = action api
        resultOfAction

    /// ('a -> 'b) -> ApiAction<'a> -> ApiAction<'b>
    let map f action =
        let newAction api =
            let x = run api action
            f x
        ApiAction newAction

    /// 'a -> ApiAction<'a>
    let retn x =
        let newAction api =
            x
        ApiAction newAction

    /// ApiAction<('a -> 'b)> -> ApiAction<'a> -> ApiAction<'b>
    let apply fAction xAction =
        let newAction api =
            let f = run api fAction
            let x = run api xAction
            f x
        ApiAction newAction

    /// ('a -> ApiAction<'b>) -> ApiAction<'a> -> ApiAction<'b>
    let bind f xAction =
        let newAction api =
            let x = run api xAction
            run api (f x)
        ApiAction newAction

    /// Create an ApiClient and run the action on it
    /// ApiAction<'a> -> 'a
    let execute action =
        use api = new ApiClient()
        api.Open()
        let result = run api action
        api.Close()
        result
```

Note that all the functions use a helper function called `run` which unwraps an `ApiAction` to get the function inside, and applies this to the `api` that is also passed in. The result is the value wrapped in the `ApiAction`.

For example, if we had an `ApiAction<int>` then `run api myAction` would result in an `int`.

And at the bottom, there is a `execute` function that creates an `ApiClient`, opens the connection, runs the action, and then closes the connection.

And with the core functions for `ApiAction` defined, we can go ahead and define the functions for the compound type `ApiActionResult`, just as we did for `AsyncResult` in the previous post:

```
module ApiActionResult =

    let map f  =
        ApiAction.map (Result.map f)

    let retn x =
        ApiAction.retn (Result.retn x)

    let apply fActionResult xActionResult =
        let newAction api =
            let fResult = ApiAction.run api fActionResult
            let xResult = ApiAction.run api xActionResult
            Result.apply fResult xResult
        ApiAction newAction

    let bind f xActionResult =
        let newAction api =
            let xResult = ApiAction.run api xActionResult
            // create a new action based on what xResult is
            let yAction =
                match xResult with
                | Success x ->
                    // Success? Run the function
                    f x
                | Failure err ->
                    // Failure? wrap the error in an ApiAction
                    (Failure err) |> ApiAction.retn
            ApiAction.run api yAction
        ApiAction newAction
```

Working out the transforms

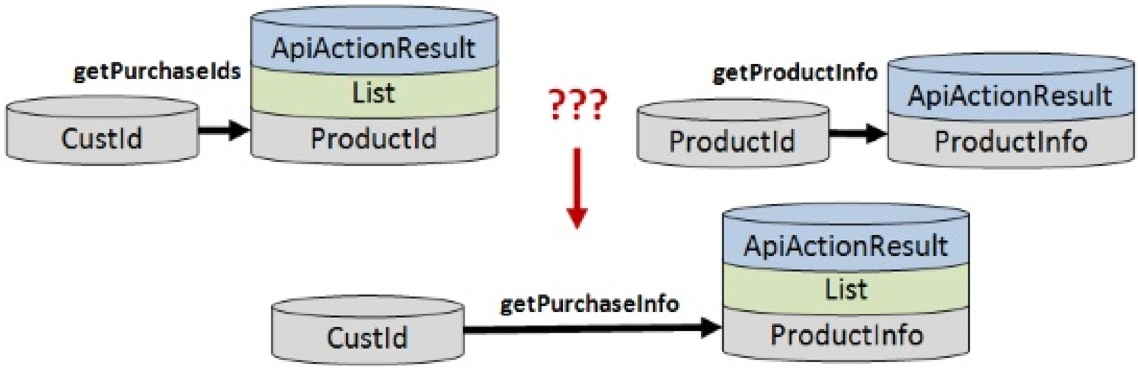
Now that we have all the tools in place, we must decide on what transforms to use to change the shape of `getProductInfo` so that the input matches up.

Should we choose `map`, or `bind`, or `traverse`?

Let's play around with the stacks visually and see what happens for each kind of transform.

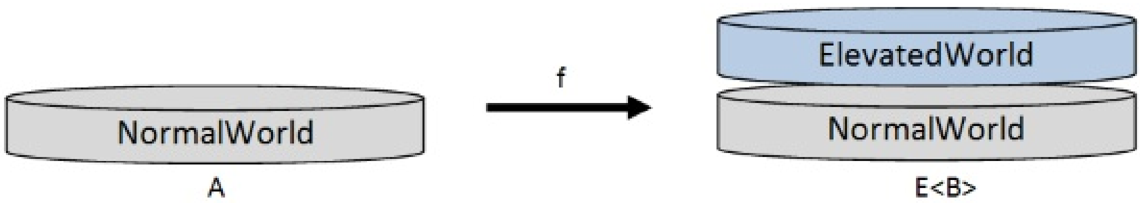
Before we get started, let's be explicit about what we are trying to achieve:

- We have two functions `getPurchaseIds` and `getProductInfo` that we want to combine into a single function `getPurchaseInfo`.
- We have to manipulate the *left* side (the input) of `getProductInfo` so that it matches the output of `getPurchaseIds`.
- We have to manipulate the *right* side (the output) of `getProductInfo` so that it matches the output of our ideal `getPurchaseInfo`.



Map

As a reminder, `map` adds a new stack on both sides. So if we start with a generic world-crossing function like this:



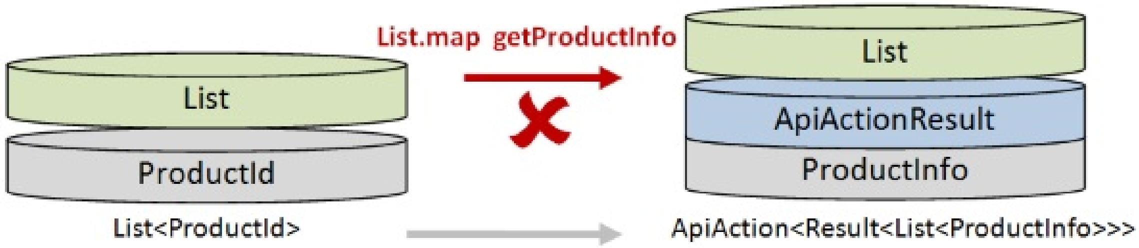
Then, after `List.map` say, we will have a new `List` stack on each site.



Here's our `getProductInfo` before transformation:



And here is what it would look like after using `List.map`



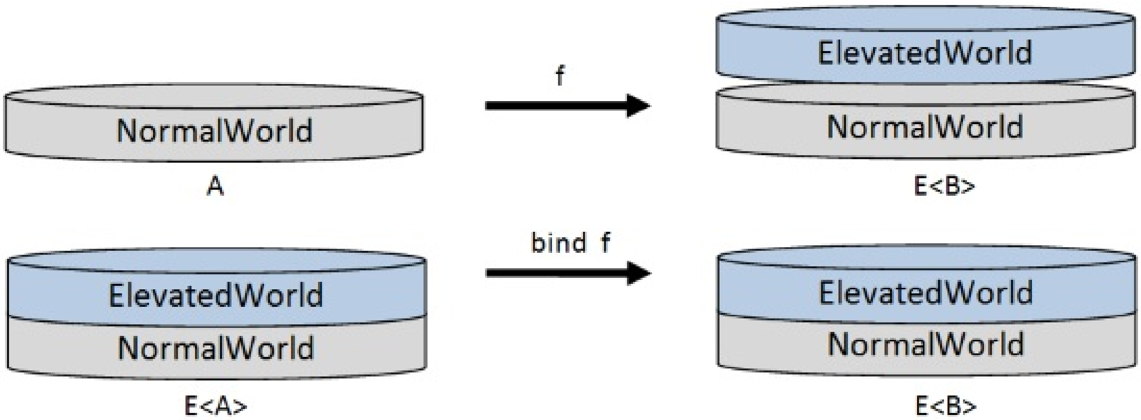
This might seem promising -- we have a `List` of `ProductId` as input now, and if we can stack a `ApiActionResult` on top we would match the output of `getPurchaseId`.

But the output is all wrong. We want the `ApiActionResult` to stay on the top. That is, we don't want a `List` of `ApiActionResult` but a `ApiActionResult` of `List`.

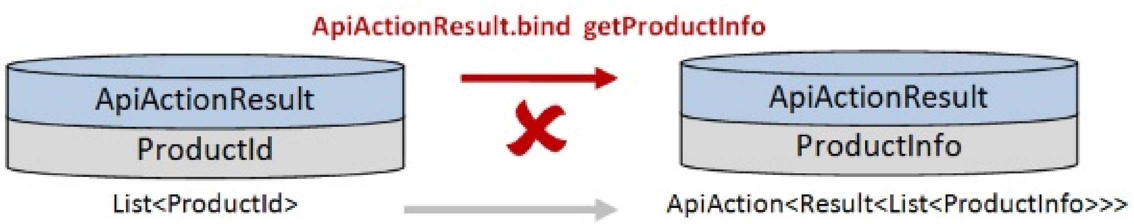
Bind

Ok, what about `bind`?

If you recall, `bind` turns a "diagonal" function into a horizontal function by adding a new stack on the *left* sides. So for example, whatever the top elevated world is on the right, that will be added to the left.



And here is what our `getProductInfo` would look like after using `ApiActionResult.bind`

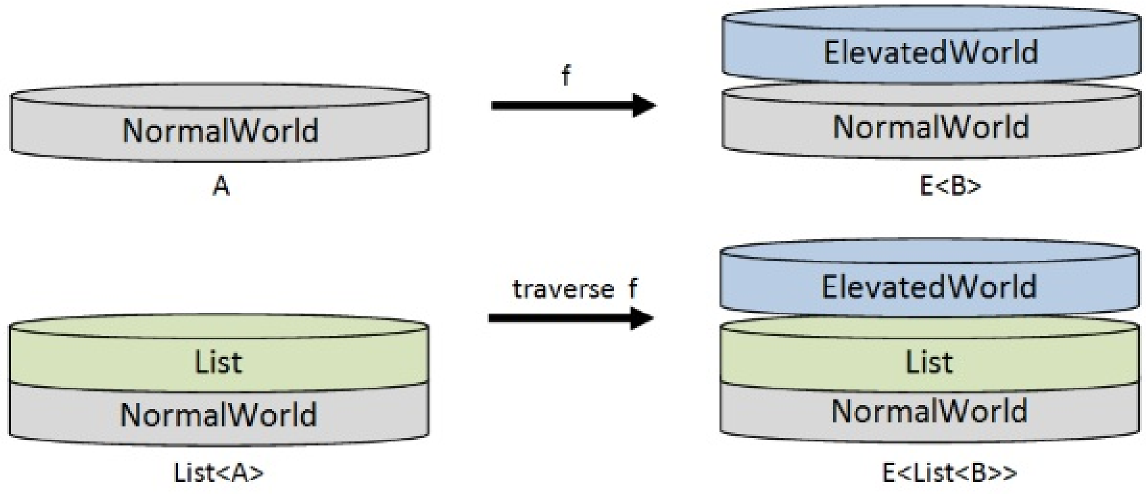


This is no good to us. We need to have a `List` of `ProductId` as input.

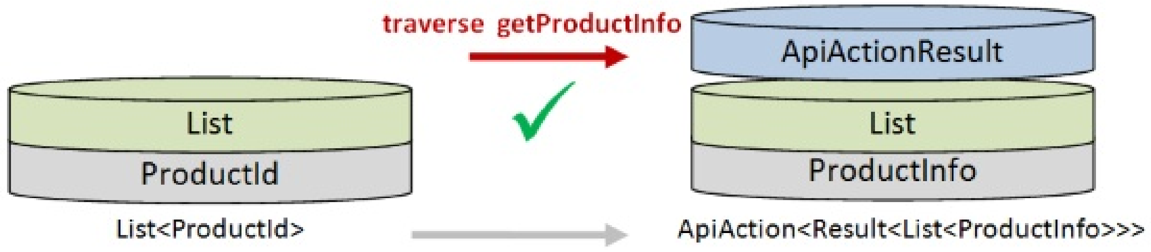
Traverse

Finally, let's try `traverse`.

`traverse` turns a diagonal function of values into diagonal function with lists wrapping the values. That is, `List` is added as the top stack on the left hand side, and the second-from-top stack on the right hand side.



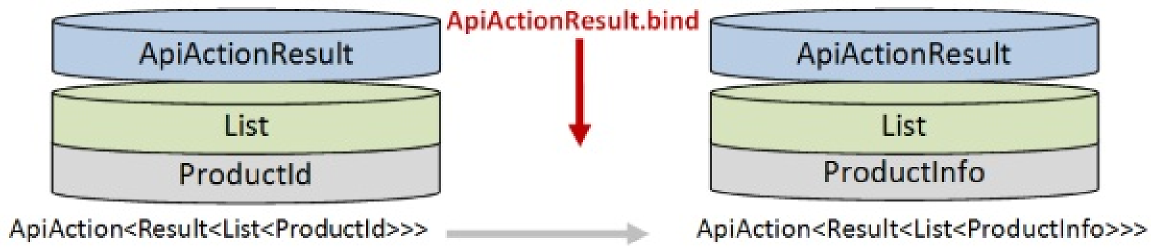
if we try that out on `getProductInfo` we get something very promising.



The input is a list as needed. And the output is perfect. We wanted a `ApiAction<Result<List<ProductInfo>>>` and we now have it.

So all we need to do now is add an `ApiActionResult` to the left side.

Well, we just saw this! It's `bind`. So if we do that as well, we are finished.



And here it is expressed as code:

```
let getPurchaseInfo =
    let getProductInfo1 = traverse getProductInfo
    let getProductInfo2 = ApiActionResult.bind getProductInfo1
    getPurchaseIds >> getProductInfo2
```

Or to make it a bit less ugly:

```
let getPurchaseInfo =
    let getProductInfoLifted =
        getProductInfo
        |> traverse
        |> ApiActionResult.bind
    getPurchaseIds >> getProductInfoLifted
```

Let's compare that with the earlier version of `getPurchaseInfo` :

```
let getPurchaseInfo (custId:CustId) (api:ApiClient) =

    let result = Result.result {
        let! productIds = getPurchaseIds custId api

        let productInfos = ResizeArray()
        for productId in productIds do
            let! productInfo = getProductInfo productId api
            productInfos.Add productInfo
        return productInfos |> List.ofSeq
        }

    // return result
    result
```

Let's compare the two versions in a table:

Earlier verson	Latest function
Composite function is non-trivial and needs special code to glue the two smaller functions together	Composite function is just piping and composition
Uses the "result" computation expression	No special syntax needed
Has special code to loop through the results	Uses "traverse"
Uses a intermediate (and mutable) List object to accumulate the list of product infos	No intermediate values needed. Just a data pipeline.

Implementing traverse

The code above uses `traverse` , but we haven't implemented it yet. As I noted earlier, it can be implemented mechanically, following a template.

Here it is:

```
let traverse f list =
    // define the applicative functions
    let (<*>) = ApiActionResult.apply
    let retn = ApiActionResult.retn

    // define a "cons" function
    let cons head tail = head :: tail

    // right fold over the list
    let initState = retn []
    let folder head tail =
        retn cons <*> f head <*> tail

    List.foldBack folder list initState
```

Testing the implementation

Let's test it!

First we need a helper function to show results:

```
let showResult result =
    match result with
    | Success (productInfoList) ->
        printfn "SUCCESS: %A" productInfoList
    | Failure errs ->
        printfn "FAILURE: %A" errs
```

Next, we need to load the API with some test data:

```
let setupTestData (api:ApiClient) =
    //setup purchases
    api.Set (CustId "C1") [ProductId "P1"; ProductId "P2"] |> ignore
    api.Set (CustId "C2") [ProductId "PX"; ProductId "P2"] |> ignore

    //setup product info
    api.Set (ProductId "P1") {ProductName="P1-Name"} |> ignore
    api.Set (ProductId "P2") {ProductName="P2-Name"} |> ignore
    // P3 missing

// setupTestData is an api-consuming function
// so it can be put in an ApiAction
// and then that apiAction can be executed
let setupAction = ApiAction setupTestData
ApiAction.execute setupAction
```

- Customer C1 has purchased two products: P1 and P2.

- Customer C2 has purchased two products: PX and P2.
- Products P1 and P2 have some info.
- Product PX does *not* have any info.

Let's see how this works out for different customer ids.

We'll start with Customer C1. For this customer we expect both product infos to be returned:

```
CustId "C1"
|> getPurchaseInfo
|> ApiAction.execute
|> showResult
```

And here are the results:

```
[API] Opening
[API] Get CustId "C1"
[API] Get ProductId "P1"
[API] Get ProductId "P2"
[API] Closing
[API] Disposing
SUCCESS: [{ProductName = "P1-Name";}; {ProductName = "P2-Name";}]
```

What happens if we use a missing customer, such as CX?

```
CustId "CX"
|> getPurchaseInfo
|> ApiAction.execute
|> showResult
```

As expected, we get a nice "key not found" failure, and the rest of the operations are skipped as soon as the key is not found.

```
[API] Opening
[API] Get CustId "CX"
[API] Closing
[API] Disposing
FAILURE: ["Key CustId "CX" not found"]
```

What about if one of the purchased products has no info? For example, customer C2 purchased PX and P2, but there is no info for PX.

```
CustId "C2"
|> getPurchaseInfo
|> ApiAction.execute
|> showResult
```

The overall result is a failure. Any bad product causes the whole operation to fail.

```
[API] Opening
[API] Get CustId "C2"
[API] Get ProductId "PX"
[API] Get ProductId "P2"
[API] Closing
[API] Disposing
FAILURE: ["Key ProductId "PX" not found"]
```

But note that the data for product P2 is fetched even though product PX failed. Why? Because we are using the applicative version of `traverse`, so every element of the list is fetched "in parallel".

If we wanted to only fetch P2 once we knew that PX existed, then we should be using monadic style instead. We already seen how to write a monadic version of `traverse`, so I leave that as an exercise for you!

Filtering out failures

In the implementation above, the `getPurchaseInfo` function failed if *any* product failed to be found. Harsh!

A real application would probably be more forgiving. Probably what should happen is that the failed products are logged, but all the successes are accumulated and returned.

How could we do this?

The answer is simple -- we just need to modify the `traverse` function to skip failures.

First, we need to create a new helper function for `ApiActionResult`. It will allow us to pass in two functions, one for the success case and one for the error case:

```
module ApiActionResult =

    let map = ...
    let retn =  ...
    let apply = ...
    let bind = ...

    let either onSuccess onFailure xActionResult =
        let newAction api =
            let xResult = ApiAction.run api xActionResult
            let yAction =
                match xResult with
                | Result.Success x -> onSuccess x
                | Result.Failure err -> onFailure err
            ApiAction.run api yAction
        ApiAction newAction
```

This helper function helps us match both cases inside a `ApiAction` without doing complicated unwrapping. We will need this for our `traverse` that skips failures.

By the way, note that `ApiActionResult.bind` can be defined in terms of `either`:

```
let bind f =
    either
        // Success? Run the function
        (fun x -> f x)
        // Failure? wrap the error in an ApiAction
        (fun err -> (Failure err) |> ApiAction.retn)
```

Now we can define our "traverse with logging of failures" function:

```
let traverseWithLog log f list =
    // define the applicative functions
    let (<*>) = ApiActionResult.apply
    let retn = ApiActionResult.retn

    // define a "cons" function
    let cons head tail = head :: tail

    // right fold over the list
    let initState = retn []
    let folder head tail =
        (f head)
        |> ApiActionResult.either
            (fun h -> retn cons <*> retn h <*> tail)
            (fun errs -> log errs; tail)
    List.foldBack folder list initState
```

The only difference between this and the previous implementation is this bit:

```
let folder head tail =
    (f head)
    |> ApiActionResult.either
        (fun h -> retn cons <*> retn h <*> tail)
        (fun errs -> log errs; tail)
```

This says that:

- If the new first element (`f head`) is a success, lift the inner value (`retn h`) and `cons` it with the tail to build a new list.
- But if the new first element is a failure, then log the inner errors (`errs`) with the passed in logging function (`log`) and just reuse the current tail. In this way, failed elements are not added to the list, but neither do they cause the whole function to fail.

Let's create a new function `getPurchasesInfoWithLog` and try it with customer C2 and the missing product PX:

```
let getPurchasesInfoWithLog =
    let log errs = printfn "SKIPPED %A" errs
    let getProductInfoLifted =
        getProductInfo
        |> traverseWithLog log
        |> ApiActionResult.bind
    getPurchaseIds >> getProductInfoLifted

CustId "C2"
|> getPurchasesInfoWithLog
|> ApiAction.execute
|> showResult
```

The result is a Success now, but only one `ProductInfo` , for P2, is returned. The log shows that PX was skipped.

```
[API] Opening
[API] Get CustId "C2"
[API] Get ProductId "PX"
SKIPPED ["Key ProductId "PX" not found"]
[API] Get ProductId "P2"
[API] Closing
[API] Disposing
SUCCESS: [{ProductName = "P2-Name";}]
```

The Reader monad

If you look closely at the `ApiResult` module, you will see that `map`, `bind`, and all the other functions do not use any information about the `api` that is passed around. We could have made it any type and those functions would still have worked.

So in the spirit of "parameterize all the things", why not make it a parameter?

That means that we could have defined `ApiAction` as follows:

```
type ApiAction<'anything,'a> = ApiAction of ('anything -> 'a)
```

But if it can be *anything*, why call it `ApiAction` any more? It could represent any set of things that depend on an object (such as an `api`) being passed in to them.

We are not the first people to discover this! This type is commonly called the `Reader` type and is defined like this:

```
type Reader<'environment,'a> = Reader of ('environment -> 'a)
```

The extra type `'environment` plays the same role that `ApiClient` did in our definition of `ApiAction`. There is some environment that is passed around as an extra parameter to all your functions, just as a `api` instance was.

In fact, we can actually define `ApiAction` in terms of `Reader` very easily:

```
type ApiAction<'a> = Reader<ApiClient,'a>
```

The set of functions for `Reader` are exactly the same as for `ApiAction`. I have just taken the code and replaced `ApiAction` with `Reader` and `api` with `environment`!

```
module Reader =

    /// Evaluate the action with a given environment
    /// 'env -> Reader<'env,'a> -> 'a
    let run environment (Reader action) =
        let resultOfAction = action environment
        resultOfAction

    /// ('a -> 'b) -> Reader<'env,'a> -> Reader<'env,'b>
    let map f action =
        let newAction environment =
            let x = run environment action
            f x
        Reader newAction

    /// 'a -> Reader<'env,'a>
    let retn x =
        let newAction environment =
            x
        Reader newAction

    /// Reader<'env,('a -> 'b)> -> Reader<'env,'a> -> Reader<'env,'b>
    let apply fAction xAction =
        let newAction environment =
            let f = run environment fAction
            let x = run environment xAction
            f x
        Reader newAction

    /// ('a -> Reader<'env,'b>) -> Reader<'env,'a> -> Reader<'env,'b>
    let bind f xAction =
        let newAction environment =
            let x = run environment xAction
            run environment (f x)
        Reader newAction
```

The type signatures are a bit harder to read now though!

The `Reader` type, plus `bind` and `return`, plus the fact that `bind` and `return` implement the monad laws, means that `Reader` is typically called "the Reader monad" .

I'm not going to delve into the Reader monad here, but I hope that you can see how it is actually a useful thing and not some bizarre ivory tower concept.

The Reader monad vs. an explicit type

Now if you like, you could replace all the `ApiAction` code above with `Reader` code, and it would work just the same. But *should* you?

Personally, I think that while understanding the concept behind the Reader monad is important and useful, I prefer the actual implementation of `ApiAction` as I defined it originally, an explicit type rather than an alias for `Reader<ApiClient,'a>`.

Why? Well, F# doesn't have typeclasses, F# doesn't have partial application of type constructors, F# doesn't have "newtype". Basically, F# isn't Haskell! I don't think that idioms that work well in Haskell should be carried over to F# directly when the language does not offer support for it.

If you understand the concepts, you can implement all the necessary transformations in a few lines of code. Yes, it's a little extra work, but the upside is less abstraction and fewer dependencies.

I would make an exception, perhaps, if your team were all Haskell experts, and the Reader monad was familiar to everyone. But for teams of different abilities, I would err on being too concrete rather than too abstract.

Summary

In this post, we worked through another practical example, created our own elevated world which made things *much* easier, and in the process, accidentally re-invented the reader monad.

If you liked this, you can see a similar practical example, this time for the State monad, in my series on "Dr Frankenfunctor and the Monadster".

The next and final post has a quick summary of the series, and some further reading.

Map and Bind and Apply, a summary

Series summary

Well, this series turned out to be longer than I originally planned. Thanks for making it to the end!

I hope that this discussion has been helpful in understanding the various function transformations like `map` and `bind`, and given you some useful techniques for dealing with world-crossing functions -- maybe even demystified the m-word a bit!

If you want to start using these kinds of functions in your own code, I hope that you can see how easy they are to write, but you should also consider using one of the excellent F# utility libraries that contain these and much more:

- **ExtCore** (source, NuGet). ExtCore provides extensions to the F# core library (FSharp.Core) and aims to help you build industrial-strength F# applications. These extensions include additional functions for modules such as Array, List, Set, and Map; immutable IntSet, IntMap, LazyList, and Queue collections; a variety of computation expressions (workflows); and "workflow collections" -- collections modules which have been adapted to work seamlessly from within workflows.
- **FSharpx.Extras** (home page). FSharpx.Extras is part of the FSharpx series of libraries. It implements several standard monads (State, Reader, Writer, Either, Continuation, Distribution), validation with applicative functors, general functions like flip, and some asynchronous programming utilities, and functions to make C# - F# interop easier.

For example, the monadic traverse `List.traverseResultM` that I implemented in this post is already available in ExtCore here.

And if you liked this series, I have posts explaining the State monad in my series on "Dr Frankenfunctor and the Monadster" and the Either monad in my talk "Railway Oriented Programming".

As I said at the very beginning, writing this up has been a learning process for me too. I am not an expert, so if I have made any errors please do let me know.

Thanks!

Series contents

Here's a list of shortcuts to the various functions mentioned in this series:

- **Part 1: Lifting to the elevated world**
  - The `map` function
  - The `return` function
  - The `apply` function
  - The `liftN` family of functions
  - The `zip` function and ZipList world
- **Part 2: How to compose world-crossing functions**
  - The `bind` function
  - List is not a monad. Option is not a monad.
- **Part 3: Using the core functions in practice**
  - Independent and dependent data
  - Example: Validation using applicative style and monadic style
  - Lifting to a consistent world
  - Kleisli world
- **Part 4: Mixing lists and elevated values**
  - Mixing lists and elevated values
  - The `traverse` / `MapM` function
  - The `sequence` function
  - "Sequence" as a recipe for ad-hoc implementations
  - Readability vs. performance
  - Dude, where's my `filter`?
- **Part 5: A real-world example that uses all the techniques**
  - Example: Downloading and processing a list of websites
  - Treating two worlds as one
- **Part 6: Designing your own elevated world**
  - Designing your own elevated world
  - Filtering out failures
  - The Reader monad
- **Part 7: Summary**
  - List of operators mentioned
  - Further reading

Appendix: List of operators mentioned

Unlike OO languages, functional programming languages are known for their strange operators, so I thought it would be helpful to document the ones that have been used in this series, with links back to the relevant discussion.

Operator	Equivalent function	Discussion
`>>`	Left-to-right composition	Not part of this series, but discussed here
`<<`	Right-to-left composition	As above
`|>`	Left-to-right piping	As above
`<|`	Right-to-left piping	As above
`<!>`	`map`	Discussed here
`<$>`	`map`	Haskell operator for map, but not a valid operator in F#, so I'm using `<!>` in this series.
`<*>`	`apply`	Discussed here
`<*`	-	One sided combiner. Discussed here
`*>`	-	One sided combiner. Discussed here
`>>=`	Left-to-right `bind`	Discussed here
`=<<`	Right-to-left `bind`	As above
`>=>`	Left-to-right Kleisli composition	Discussed here
`<=<`	Right-to-left Kleisli composition	As above

Appendix: Further reading

Alternative tutorials:

- You Could Have Invented Monads! (And Maybe You Already Have).
- Functors, Applicatives and Monads in pictures.
- Kleisli composition ? la Up-Goer Five. I think this one is funny.
- Eric Lippert's series on monads in C#.

For the academically minded:

- Monads for Functional Programming (PDF), by Philip Wadler. One of the first monad

papers.

- Applicative Programming with Effects (PDF), by Conor McBride and Ross Paterson.
- The Essence of the Iterator Pattern (PDF), by Jeremy Gibbons and Bruno Oliveira.

F# examples:

- F# ExtCore and FSharpx.Extras have lots of useful code.
- FSharpx.Async has `map`, `apply`, `liftN` (called "Parallel"), `bind`, and other useful extensions for `Async`.
- Applicatives are very well suited for parsing, as explained in these posts:
    - Parsing with applicative functors in F#.
    - Dive into parser combinators: parsing search queries with F# and FParsec in Kiln.

In this series, we'll look at recursive types and how to use them, and on the way, we'll look at catamorphisms, tail recursion, the difference between left and right folds, and more.

- Introduction to recursive types. Don't fear the catamorphism....
- Catamorphism examples. Applying the rules to other domains.
- Introducing Folds. Threading state through a recursive data structure.
- Understanding Folds. Recursion vs. iteration.
- Generic recursive types. Implementing a domain in three ways.
- Trees in the real world. Examples using databases, JSON and error handling.

Introduction to recursive types

In this series, we'll look at recursive types and how to use them, and on the way, we'll look at catamorphisms, tail recursion, the difference between left and right folds, and more.

Series contents

Here's the contents of this series:

- **Part 1: Introduction to recursive types and catamorphisms**
  - A simple recursive type
  - Parameterize all the things
  - Introducing catamorphisms
  - Benefits of catamorphisms
  - Rules for creating a catamorphism
- **Part 2: Catamorphism examples**
  - Catamorphism example: File system domain
  - Catamorphism example: Product domain
- **Part 3: Introducing folds**
  - A flaw in our catamorphism implementation
  - Introducing `fold`
  - Problems with fold
  - Using functions as accumulators
  - Introducing `foldback`
  - Rules for creating a fold
- **Part 4: Understanding folds**
  - Iteration vs. recursion
  - Fold example: File system domain
  - Common questions about "fold"
- **Part 5: Generic recursive types**
  - LinkedList: A generic recursive type
  - Making the Gift domain generic
  - Defining a generic Container type
  - A third way to implement the gift domain
  - Abstract or concrete? Comparing the three designs
- **Part 6: Trees in the real world**
  - Defining a generic Tree type
  - The Tree type in the real world

- Mapping the Tree type
- Example: Creating a directory listing
- Example: A parallel grep
- Example: Storing the file system in a database
- Example: Serializing a Tree to JSON
- Example: Deserializing a Tree from JSON
- Example: Deserializing a Tree from JSON - with error handling

A basic recursive type

Let's start with a simple example -- how to model a gift.

As it happens, I'm a very lazy gift-giver. I always give people a book or chocolates. I generally wrap them, and sometimes, if I'm feeling particularly extravagant, I put them in a gift box and add a card too.

Let's see how I might model this in types:

```
type Book = {title: string; price: decimal}

type ChocolateType = Dark | Milk | SeventyPercent
type Chocolate = {chocType: ChocolateType ; price: decimal}

type WrappingPaperStyle =
    | HappyBirthday
    | HappyHolidays
    | SolidColor

type Gift =
    | Book of Book
    | Chocolate of Chocolate
    | Wrapped of Gift * WrappingPaperStyle
    | Boxed of Gift
    | WithACard of Gift * message:string
```

You can see that three of the cases are "containers" that refer to another `Gift`. The `Wrapped` case has some paper and a inner gift, as does the `Boxed` case, as does the `WithACard` case. The two other cases, `Book` and `Chocolate`, do not refer to a gift and can be considered "leaf" nodes or terminals.

The presence of a reference to an inner `Gift` in those three cases makes `Gift` a *recursive type*. Note that, unlike functions, the `rec` keyword is *not* needed for defining recursive types.

Let's create some example values:

```
// a Book
let wolfHall = {title="Wolf Hall"; price=20m}

// a Chocolate
let yummyChoc = {chocType=SeventyPercent; price=5m}

// A Gift
let birthdayPresent = WithACard (Wrapped (Book wolfHall, HappyBirthday), "Happy Birthd
ay")
//  WithACard (
//    Wrapped (
//      Book {title = "Wolf Hall"; price = 20M},
//      HappyBirthday),
//    "Happy Birthday")

// A Gift
let christmasPresent = Wrapped (Boxed (Chocolate yummyChoc), HappyHolidays)
//  Wrapped (
//    Boxed (
//      Chocolate {chocType = SeventyPercent; price = 5M}),
//    HappyHolidays)
```

Before we start working with these values, a word of advice...

Guideline: Avoid infinitely recursive types

I suggest that, in F#, every recursive type should consist of a mix of recursive and non-recursive cases. If there were no non-recursive elements, such as `Book`, all values of the type would have to be infinitely recursive.

For example, in the `ImpossibleGift` type below, all the cases are recursive. To construct any one of the cases you need an inner gift, and that needs to be constructed too, and so on.

```
type ImpossibleGift =
    | Boxed of ImpossibleGift
    | WithACard of ImpossibleGift * message:string
```

It is possible to create such types if you allow laziness, mutation, or reflection. But in general, in a non-lazy language like F#, it's a good idea to avoid such types.

Working with recursive types

End of public service announcement -- let's get coding!

First, say that we want a description of the gift. The logic will be:

- For the two non-recursive cases, return a string describing that case.
- For the three recursive cases, return a string that describes the case, but also includes the description of the inner gift. This means that `description` function is going to refer to itself, and therefore it must be marked with the `rec` keyword.

Here's an example implementation:

```
let rec description gift =
    match gift with
    | Book book ->
        sprintf "'%s'" book.title
    | Chocolate choc ->
        sprintf "%A chocolate" choc.chocType
    | Wrapped (innerGift,style) ->
        sprintf "%s wrapped in %A paper" (description innerGift) style
    | Boxed innerGift ->
        sprintf "%s in a box" (description innerGift)
    | WithACard (innerGift,message) ->
        sprintf "%s with a card saying '%s'" (description innerGift) message
```

Note the recursive calls like this one in the `Boxed` case:

```
    | Boxed innerGift ->
        sprintf "%s in a box" (description innerGift)
                               ~~~~~~~~~~~ <= recursive call
```

If we try this with our example values, let's see what we get:

```
birthdayPresent |> description
// "'Wolf Hall' wrapped in HappyBirthday paper with a card saying 'Happy Birthday'"

christmasPresent |> description
// "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"
```

That looks pretty good to me. Things like `HappyHolidays` look a bit funny without spaces, but it's good enough to demonstrate the idea.

What about creating another function? For example, what is the total cost of a gift?

For `totalCost`, the logic will be:

- Books and chocolate capture the price in the case-specific data, so use that.
- Wrapping adds `0.5` to the cost.
- A box adds `1.0` to the cost.

- A card adds `2.0` to the cost.

```
let rec totalCost gift =
    match gift with
    | Book book ->
        book.price
    | Chocolate choc ->
        choc.price
    | Wrapped (innerGift,style) ->
        (totalCost innerGift) + 0.5m
    | Boxed innerGift ->
        (totalCost innerGift) + 1.0m
    | WithACard (innerGift,message) ->
        (totalCost innerGift) + 2.0m
```

And here are the costs for the two examples:

```
birthdayPresent |> totalCost
// 22.5m

christmasPresent |> totalCost
// 6.5m
```

Sometimes, people ask what is inside the box or wrapping paper. A `whatsInside` function is easy to implement -- just ignore the container cases and return something for the non-recursive cases.

```
let rec whatsInside gift =
    match gift with
    | Book book ->
        "A book"
    | Chocolate choc ->
        "Some chocolate"
    | Wrapped (innerGift,style) ->
        whatsInside innerGift
    | Boxed innerGift ->
        whatsInside innerGift
    | WithACard (innerGift,message) ->
        whatsInside innerGift
```

And the results:

```
birthdayPresent |> whatsInside
// "A book"

christmasPresent |> whatsInside
// "Some chocolate"
```

So that's a good start -- three functions, all quite easy to write.

Parameterize all the things

However, these three functions have some duplicate code. In addition to the unique application logic, each function is doing its own pattern matching and its own logic for recursively visiting the inner gift.

How can we separate the navigation logic from the application logic?

Answer: Parameterize all the things!

As always, we can parameterize the application logic by passing in functions. In this case, we will need *five* functions, one for each case.

Here is the new, parameterized version -- I'll explain why I have called it `cataGift` shortly.

```
let rec cataGift fBook fChocolate fWrapped fBox fCard gift =
    match gift with
    | Book book ->
        fBook book
    | Chocolate choc ->
        fChocolate choc
    | Wrapped (innerGift,style) ->
        let innerGiftResult = cataGift fBook fChocolate fWrapped fBox fCard innerGift
        fWrapped (innerGiftResult,style)
    | Boxed innerGift ->
        let innerGiftResult = cataGift fBook fChocolate fWrapped fBox fCard innerGift
        fBox innerGiftResult
    | WithACard (innerGift,message) ->
        let innerGiftResult = cataGift fBook fChocolate fWrapped fBox fCard innerGift
        fCard (innerGiftResult,message)
```

You can see this function is created using a purely mechanical process:

- Each function parameter (`fBook` , `fChocolate` , etc) corresponds to a case.
- For the two non-recursive cases, the function parameter is passed all the data associated with that case.
- For the three recursive cases, there are two steps:
    - First, the `cataGift` function is called recursively on the `innerGift` to get an `innerGiftResult`
    - Then the appropriate handler is passed all the data associated with that case, but with `innerGiftResult` replacing `innerGift` .

Let's rewrite total cost using the generic `cataGift` function.

```
let totalCostUsingCata gift =
    let fBook (book:Book) =
        book.price
    let fChocolate (choc:Chocolate) =
        choc.price
    let fWrapped  (innerCost,style) =
        innerCost + 0.5m
    let fBox innerCost =
        innerCost + 1.0m
    let fCard (innerCost,message) =
        innerCost + 2.0m
    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

Notes:

- The `innerGiftResult` is now the total cost of the inner gift, so I have renamed it to `innerCost`.
- The `totalCostUsingCata` function itself is not recursive, because it uses the `cataGift` function, and so no longer needs the `rec` keyword.

And this function gives the same result as before:

```
birthdayPresent |> totalCostUsingCata
// 22.5m
```

We can rewrite the `description` function using `cataGift` in the same way, changing `innerGiftResult` to `innerText`.

```
let descriptionUsingCata gift =
    let fBook (book:Book) =
        sprintf "'%s'" book.title
    let fChocolate (choc:Chocolate) =
        sprintf "%A chocolate" choc.chocType
    let fWrapped (innerText,style) =
        sprintf "%s wrapped in %A paper" innerText style
    let fBox innerText =
        sprintf "%s in a box" innerText
    let fCard (innerText,message) =
        sprintf "%s with a card saying '%s'" innerText message
    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

And the results are the same as before:

```
birthdayPresent |> descriptionUsingCata
// "'Wolf Hall' wrapped in HappyBirthday paper with a card saying 'Happy Birthday'"

christmasPresent |> descriptionUsingCata
// "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"
```

Introducing catamorphisms

The `cataGift` function we wrote above is called a "catamorphism", from the Greek components "down + shape". In normal usage, a catamorphism is a function that "collapses" a recursive type into a new value based on its *structure*. In fact, you can think of a catamorphism as a sort of "visitor pattern".

A catamorphism is very powerful concept, because it is the most fundamental function that you can define for a structure like this. *Any other function* can be defined in terms of it.

That is, if we want to create a function with signature `Gift -> string` or `Gift -> int`, we can use a catamorphism to create it by specifying a function for each case in the `Gift` structure.

We saw above how we could rewrite `totalCost` as `totalCostUsingCata` using the catamorphism, and we'll see lots of other examples later.

Catamorphisms and folds

Catamorphisms are often called "folds", but there is more than one kind of fold, so I will tend to use "catamorphism" to refer the *concept* and "fold" to refer to a specific kind of implementation.

I will talk in detail about the various kinds of folds in a subsequent post, so for the rest of this post I will just use "catamorphism".

Tidying up the implementation

The `cataGift` implementation above was deliberately long-winded so that you could see each step. Once you understand what is going on though, you can simplify it somewhat.

First, the `cataGift fBook fChocolate fWrapped fBox fCard` crops up three times, once for each recursive case. Let's assign it a name like `recurse`:

```
let rec cataGift fBook fChocolate fWrapped fBox fCard gift =
    let recurse = cataGift fBook fChocolate fWrapped fBox fCard
    match gift with
    | Book book ->
        fBook book
    | Chocolate choc ->
        fChocolate choc
    | Wrapped (innerGift,style) ->
        let innerGiftResult = recurse innerGift
        fWrapped (innerGiftResult,style)
    | Boxed innerGift ->
        let innerGiftResult = recurse innerGift
        fBox innerGiftResult
    | WithACard (innerGift,message) ->
        let innerGiftResult = recurse innerGift
        fCard (innerGiftResult,message)
```

The `recurse` function has the simple signature `Gift -> 'a` -- that is, it converts a `Gift` to the return type we need, and so we can use it to work with the various `innerGift` values.

The other thing is to replace `innerGift` with just `gift` in the recursive cases -- this is called "shadowing". The benefit is that the "outer" `gift` is no longer visible to the case-handling code, and so we can't accidentally recurse into it, which would cause an infinite loop.

Generally I avoid shadowing, but this is one case where it actually is a good practice, because it eliminates a particularly nasty kind of bug.

Here's the version after the clean up:

```
let rec cataGift fBook fChocolate fWrapped fBox fCard gift =
    let recurse = cataGift fBook fChocolate fWrapped fBox fCard
    match gift with
    | Book book ->
        fBook book
    | Chocolate choc ->
        fChocolate choc
    | Wrapped (gift,style) ->
        fWrapped (recurse gift,style)
    | Boxed gift ->
        fBox (recurse gift)
    | WithACard (gift,message) ->
        fCard (recurse gift,message)
```

One more thing. I'm going to explicitly annotate the return type and call it `'r`. Later on in this series we'll be dealing with other generic types such as `'a` and `'b`, so it will be helpful to be consistent and always have a standard name for the return type.

```
let rec cataGift fBook fChocolate fWrapped fBox fCard gift :'r =
//                                name the return type =>  ~~~~
```

Here's the final version:

```
let rec cataGift fBook fChocolate fWrapped fBox fCard gift :'r =
    let recurse = cataGift fBook fChocolate fWrapped fBox fCard
    match gift with
    | Book book ->
        fBook book
    | Chocolate choc ->
        fChocolate choc
    | Wrapped (gift,style) ->
        fWrapped (recurse gift,style)
    | Boxed gift ->
        fBox (recurse gift)
    | WithACard (gift,message) ->
        fCard (recurse gift,message)
```

It's much simpler than the original implementation, and also demonstrates the symmetry between a case constructor like `Wrapped (gift,style)` and the corresponding handler `fWrapped (recurse gift,style)`. Which leads us nicely to...

The relationship between the type constructors and the handlers

Here is the signature for the `cataGift` function. You can see that each case handler function (`fBook` , `fBox` , etc.) has the same pattern: an input which contains all the data for that case, and a common output type `'r` .

```
val cataGift :
  fBook:(Book -> 'r) ->
  fChocolate:(Chocolate -> 'r) ->
  fWrapped:('r * WrappingPaperStyle -> 'r) ->
  fBox:('r -> 'r) ->
  fCard:('r * string -> 'r) ->
  // input value
  gift:Gift ->
  // return value
  'r
```

Another way to think about this is that: everywhere that there is a `Gift` type in the constructor, it has been replaced with an `'r` .

For example:

- The `Gift.Book` constructor takes a `Book` and returns a `Gift`. The `fBook` handler takes a `Book` and returns an `'r`.
- The `Gift.Wrapped` constructor takes a `Gift * WrappingPaperStyle` and returns a `Gift`. The `fWrapped` handler takes an `'r * WrappingPaperStyle` as input and returns an `'r`.

Here is that relationship expressed through type signatures:

```
// The Gift.Book constructor
Book -> Gift

// The fBook handler
Book -> 'r

// The Gift.Wrapped constructor
Gift * WrappingPaperStyle -> Gift

// The fWrapped handler
'r   * WrappingPaperStyle -> 'r

// The Gift.Boxed constructor
Gift -> Gift

// The fBox handler
'r   -> 'r
```

and so on for all of the rest.

Benefits of catamorphisms

There is a lot of theory behind catamorphisms, but what are the benefits in practice?

Why bother to create a special function like `cataGift`? Why not just leave the original functions alone?

There are number of reasons, including:

- **Reuse**. Later, we will be creating quite complex catamorphisms. It's nice if you only have to get the logic right once.
- **Encapsulation**. By exposing functions only, you hide the internal structure of the data type.
- **Flexibility**. Functions are more flexible than pattern matching -- they can be composed, partially applied, etc.
- **Mapping**. With a catamorphism in hand you can easily create functions that map the various cases to new structures.

It's true that most of these benefits apply to non-recursive types as well, but recursive types tend to be more complex so the benefits of encapsulation, flexibility, etc. are correspondingly stronger.

In the following sections, we'll look at the last three points in more detail.

Using function parameters to hide internal structure

The first benefit is that a catamorphism abstracts out the internal design. By using functions, the client code is somewhat isolated from the internal structure. Again, you can think of the Visitor pattern as analogous in the OO world.

For example, if all the clients used the catamorphism function rather than pattern matching, I could safely rename cases, and even, with a bit of care, add or remove cases.

Here's an example. Let's say that I had a earlier design for `Gift` that didn't have the `WithACard` case. I'll call it version 1:

```
type Gift =
    | Book of Book
    | Chocolate of Chocolate
    | Wrapped of Gift * WrappingPaperStyle
    | Boxed of Gift
```

And say that we built and published a catamorphism function for that structure:

```
let rec cataGift fBook fChocolate fWrapped fBox gift :'r =
    let recurse = cataGift fBook fChocolate fWrapped fBox
    match gift with
    | Book book ->
        fBook book
    | Chocolate choc ->
        fChocolate choc
    | Wrapped (gift,style) ->
        fWrapped (recurse gift,style)
    | Boxed gift ->
        fBox (recurse gift)
```

Note that this has only *four* function parameters.

Now suppose that version 2 of `Gift` comes along, which adds the `WithACard` case:

```
type Gift =
    | Book of Book
    | Chocolate of Chocolate
    | Wrapped of Gift * WrappingPaperStyle
    | Boxed of Gift
    | WithACard of Gift * message:string
```

and now there are five cases.

Often, when we add a new case, we *want* to break all the clients and force them to deal with the new case.

But sometimes, we don't. And so we can stay compatible with the original `cataGift` by handling the extra case silently, like this:

```
/// Uses Gift_V2 but is still backwards compatible with the earlier "cataGift".
let rec cataGift fBook fChocolate fWrapped fBox gift :'r =
    let recurse = cataGift fBook fChocolate fWrapped fBox
    match gift with
    | Book book ->
        fBook book
    | Chocolate choc ->
        fChocolate choc
    | Wrapped (gift,style) ->
        fWrapped (recurse gift,style)
    | Boxed gift ->
        fBox (recurse gift)
    // pass through the new case silently
    | WithACard (gift,message) ->
        recurse gift
```

This function still has only four function parameters -- there is no special behavior for the `WithACard` case.

There are a number of alternative ways of being compatible, such as returning a default value. The important point is that the clients are not aware of the change.

Aside: Using active patterns to hide data

While we're on the topic of hiding the structure of a type, I should mention that you can also use active patterns to do this.

For example, we could create a active pattern for the first four cases, and ignore the `WithACard` case.

```
let rec (|Book|Chocolate|Wrapped|Boxed|) gift =
    match gift with
    | Gift.Book book ->
        Book book
    | Gift.Chocolate choc ->
        Chocolate choc
    | Gift.Wrapped (gift,style) ->
        Wrapped (gift,style)
    | Gift.Boxed gift ->
        Boxed gift
    | Gift.WithACard (gift,message) ->
        // ignore the message and recurse into the gift
        (|Book|Chocolate|Wrapped|Boxed|) gift
```

The clients can pattern match on the four cases without knowing that the new case even exists:

```
let rec whatsInside gift =
    match gift with
    | Book book ->
        "A book"
    | Chocolate choc ->
        "Some chocolate"
    | Wrapped (gift,style) ->
        whatsInside gift
    | Boxed gift ->
        whatsInside gift
```

Case-handling functions vs. pattern matching

Catamorphisms use function parameters, and as noted above, functions are more flexible than pattern matching due to tools such composition, partial application, etc.

Here's an example where all the "container" cases are ignored, and only the "content" cases are handled.

```
let handleContents fBook fChocolate gift =
    let fWrapped (innerGiftResult,style) =
        innerGiftResult
    let fBox innerGiftResult =
        innerGiftResult
    let fCard (innerGiftResult,message) =
        innerGiftResult

    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

And here it is in use, with the two remaining cases handled "inline" using piping:

```
birthdayPresent
|> handleContents
    (fun book -> "The book you wanted for your birthday")
    (fun choc -> "Your fave chocolate")
// Result => "The book you wanted for your birthday"

christmasPresent
|> handleContents
    (fun book -> "The book you wanted for Christmas")
    (fun choc -> "Don't eat too much over the holidays!")
// Result => "Don't eat too much over the holidays!"
```

Of course this could be done with pattern matching, but being able to work with the existing `cataGift` function directly makes life easier.

Using catamorphisms for mapping

I said above that a catamorphism is a function that "collapses" a recursive type into a new value. For example, in `totalCost`, the recursive gift structure was collapsed into a single cost value.

But a "single value" can be more than just a primitive -- it can be a complicated structure too, such as a another recursive structure.

In fact, catamorphisms are great for mapping one kind of structure onto another, especially if they are very similar.

For example, let's say that I have a chocolate-loving room-mate who will stealthily remove and devour any chocolate in a gift, leaving the wrapping untouched, but discarding the box and gift card.

What's left at the end is a "gift minus chocolate" that we can model as follows:

```
type GiftMinusChocolate =
    | Book of Book
    | Apology of string
    | Wrapped of GiftMinusChocolate * WrappingPaperStyle
```

We can easily map from a `Gift` to a `GiftMinusChocolate`, because the cases are almost parallel.

- A `Book` is passed through untouched.
- `Chocolate` is eaten and replaced with an `Apology`.
- The `Wrapped` case is passed through untouched.

- The `Box` and `WithACard` cases are ignored.

Here's the code:

```
let removeChocolate gift =
    let fBook (book:Book) =
        Book book
    let fChocolate (choc:Chocolate) =
        Apology "sorry I ate your chocolate"
    let fWrapped (innerGiftResult,style) =
        Wrapped (innerGiftResult,style)
    let fBox innerGiftResult =
        innerGiftResult
    let fCard (innerGiftResult,message) =
        innerGiftResult
    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

And if we test...

```
birthdayPresent |> removeChocolate
// GiftMinusChocolate =
//     Wrapped (Book {title = "Wolf Hall"; price = 20M}, HappyBirthday)

christmasPresent |> removeChocolate
// GiftMinusChocolate =
//     Wrapped (Apology "sorry I ate your chocolate", HappyHolidays)
```

Deep copying

One more thing. Remember that each case-handling function takes the data associated with that case? That means that we can just use *the original case constructors* as the functions!

To see what I mean, let's define a function called `deepCopy` that clones the original value. Each case handler is just the corresponding case constructor:

```
let deepCopy gift =
    let fBook book =
        Book book
    let fChocolate (choc:Chocolate) =
        Chocolate choc
    let fWrapped (innerGiftResult,style) =
        Wrapped (innerGiftResult,style)
    let fBox innerGiftResult =
        Boxed innerGiftResult
    let fCard (innerGiftResult,message) =
        WithACard (innerGiftResult,message)
    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

We can simplify this further by removing the redundant parameters for each handler:

```
let deepCopy gift =
    let fBook = Book
    let fChocolate = Chocolate
    let fWrapped = Wrapped
    let fBox = Boxed
    let fCard = WithACard
    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

You can test that this works yourself:

```
christmasPresent |> deepCopy
// Result =>
//   Wrapped (
//    Boxed (Chocolate {chocType = SeventyPercent; price = 5M;}),
//    HappyHolidays)
```

So that leads to another way of thinking about a catamorphism:

- A catamorphism is a function for a recursive type such that when you pass in the type's case constructors, you get a "clone" function.

Mapping and transforming in one pass

A slight variant on `deepCopy` allows us to recurse through an object and change bits of it as we do so.

For example, let's say I don't like milk chocolate. Well, I can write a function that upgrades the gift to better quality chocolate and leaves all the other cases alone.

```
let upgradeChocolate gift =
    let fBook = Book
    let fChocolate (choc:Chocolate) =
        Chocolate {choc with chocType = SeventyPercent}
    let fWrapped = Wrapped
    let fBox = Boxed
    let fCard = WithACard
    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

And here it is in use:

```
// create some chocolate I don't like
let cheapChoc = Boxed (Chocolate {chocType=Milk; price=5m})

// upgrade it!
cheapChoc |> upgradeChocolate
// Result =>
//    Boxed (Chocolate {chocType = SeventyPercent; price = 5M})
```

If you are thinking that this is beginning to smell like a `map`, you'd be right. We'll look at generic maps in the sixth post, as part of a discussion of generic recursive types.

Rules for creating catamorphisms

We saw above that creating a catamorphism is a mechanical process:

- Create a function parameter to handle each case in the structure.
- For non-recursive cases, pass the function parameter all the data associated with that case.
- For recursive cases, perform two steps:
  - First, call the catamorphism recursively on the nested value.
  - Then pass the handler all the data associated with that case, but with the result of the catamorphism replacing the original nested value.

Let's now see if we can apply these rules to create catamorphisms in other domains.

Summary

We've seen in this post how to define a recursive type, and been introduced to catamorphisms.

In the next post we'll uses these rules to create catamorphisms for some other domains.

See you then!

The source code for this post is available at this gist.

Catamorphism examples

This post is the second in a series.

In the previous post, I introduced "catamorphisms", a way of creating functions for recursive types, and listed some rules which can be used to implement them mechanically. In this post, we'll use these rules to implement catamorphisms for some other domains.

Series contents

Here's the contents of this series:

- **Part 1: Introduction to recursive types and catamorphisms**
  - A simple recursive type
  - Parameterize all the things
  - Introducing catamorphisms
  - Benefits of catamorphisms
  - Rules for creating a catamorphism
- **Part 2: Catamorphism examples**
  - Catamorphism example: File system domain
  - Catamorphism example: Product domain
- **Part 3: Introducing folds**
  - A flaw in our catamorphism implementation
  - Introducing `fold`
  - Problems with fold
  - Using functions as accumulators
  - Introducing `foldback`
  - Rules for creating a fold
- **Part 4: Understanding folds**
  - Iteration vs. recursion
  - Fold example: File system domain
  - Common questions about "fold"
- **Part 5: Generic recursive types**
  - LinkedList: A generic recursive type
  - Making the Gift domain generic
  - Defining a generic Container type
  - A third way to implement the gift domain
  - Abstract or concrete? Comparing the three designs

- **Part 6: Trees in the real world**
  - Defining a generic Tree type
  - The Tree type in the real world
  - Mapping the Tree type
  - Example: Creating a directory listing
  - Example: A parallel grep
  - Example: Storing the file system in a database
  - Example: Serializing a Tree to JSON
  - Example: Deserializing a Tree from JSON
  - Example: Deserializing a Tree from JSON - with error handling

---

Rules for creating catamorphisms

We saw in the previous post that creating a catamorphism is a mechanical process, and the rules were:

- Create a function parameter to handle each case in the structure.
- For non-recursive cases, pass the function parameter all the data associated with that case.
- For recursive cases, perform two steps:
  - First, call the catamorphism recursively on the nested value.
  - Then pass the handler all the data associated with that case, but with the result of the catamorphism replacing the original nested value.

Let's now see if we can apply these rules to create catamorphisms in other domains.

Catamorphism example: File system domain

Let's start with a very crude model of a file system:

- Each file has a name and a size.
- Each directory has a name and a size and a list of subitems.

Here's how I might model that:

```
type FileSystemItem =
    | File of File
    | Directory of Directory
and File = {name:string; fileSize:int}
and Directory = {name:string; dirSize:int; subitems:FileSystemItem list}
```

I admit it's a pretty bad model, but it's just good enough for this example!

Ok, here are some sample files and directories:

```
let readme = File {name="readme.txt"; fileSize=1}
let config = File {name="config.xml"; fileSize=2}
let build  = File {name="build.bat"; fileSize=3}
let src = Directory {name="src"; dirSize=10; subitems=[readme; config; build]}
let bin = Directory {name="bin"; dirSize=10; subitems=[]}
let root = Directory {name="root"; dirSize=5; subitems=[src; bin]}
```

Time to create the catamorphism!

Let's start by looking at the signatures to figure out what we need.

The `File` constructor takes a `File` and returns a `FileSystemItem`. Using the guidelines above, the handler for the `File` case needs to have the signature `File -> 'r`.

```
// case constructor
File  : File -> FileSystemItem

// function parameter to handle File case
fFile : File -> 'r
```

That's simple enough. Let's put together an initial skeleton of `cataFS`, as I'll call it:

```
let rec cataFS fFile fDir item :'r =
    let recurse = cataFS fFile fDir
    match item with
    | File file ->
        fFile file
    | Directory dir ->
        // to do
```

The `Directory` case is more complicated. If we naively applied the guidelines above, the handler for the `Directory` case would have the signature `Directory -> 'r`, but that would be incorrect, because the `Directory` record itself contains a `FileSystemItem` that needs to be replaced with an `'r` too. How can we do this?

One way is to "explode" the `Directory` record into a tuple of `(string,int,FileSystemItem list)`, and then replace the `FileSystemItem` with `'r` in there too.

In other words, we have this sequence of transformations:

```
// case constructor (Directory as record)
Directory : Directory -> FileSystemItem

// case constructor (Directory unpacked as tuple)
Directory : (string, int, FileSystemItem list) -> FileSystemItem
//   replace with 'r ===> ~~~~~~~~~~~~~~          ~~~~~~~~~~~~~~

// function parameter to handle Directory case
fDir :      (string, int, 'r list)             -> 'r
```

Another issue is that the data associated with the Directory case is a *list* of `FileSystemItem` s. How can we convert that into a list of `'r` s?

Well, the `recurse` helper turns a `FileSystemItem` into an `'r`, so we can just use `List.map` passing in `recurse` as the mapping function, and that will give us the list of `'r` s we need!

Putting it all together, we get this implementation:

```
let rec cataFS fFile fDir item :'r =
    let recurse = cataFS fFile fDir
    match item with
    | File file ->
        fFile file
    | Directory dir ->
        let listOfRs = dir.subitems |> List.map recurse
        fDir (dir.name,dir.dirSize,listOfRs)
```

and if we look at the type signature, we can see that it is just what we want:

```
val cataFS :
    fFile : (File -> 'r) ->
    fDir  : (string * int * 'r list -> 'r) ->
    // input value
    FileSystemItem ->
    // return value
    'r
```

So we're done. It's a bit complicated to set up, but once built, we have a nice reusable function that can be the basis for many others.

File system domain: `totalSize` example

Alrighty then, let's use it in practice.

To start with, we can easily define a `totalSize` function that returns the total size of an item and all its subitems:

```
let totalSize fileSystemItem =
    let fFile (file:File) =
        file.fileSize
    let fDir (name,size,subsizes) =
        (List.sum subsizes) + size
    cataFS fFile fDir fileSystemItem
```

And here are the results:

```
readme |> totalSize  // 1
src |> totalSize     // 16 = 10 + (1 + 2 + 3)
root |> totalSize    // 31 = 5 + 16 + 10
```

File system domain: `largestFile` example

How about a more complicated function, such as "what is the largest file in the tree?"

Before we start this one, let's think about what it should return. That is, what is the `'r` ?

You might think that it's just a `File` . But what if the subdirectory is empty and there *are* no files?

So let's make `'r` a `File option` instead.

The function for the `File` case should return `Some file` then:

```
let fFile (file:File) =
    Some file
```

The function for the `Directory` case needs more thought:

- If list of subfiles is empty, then return `None`
- If list of subfiles is non-empty, then return the largest one

```
let fDir (name,size,subfiles) =
    match subfiles with
    | [] ->
        None  // empty directory
    | subfiles ->
        // return largest one
```

But remember that `'r` is not a `File` but a `File option`. So that means that `subfiles` is not a list of files, but a list of `File option`.

Now, how can we find the largest one of these? We probably want to use `List.maxBy` and pass in the size. But what is the size of a `File option`?

Let's write a helper function to provide the size of a `File option`, using this logic:

- If the `File option` is `None`, return 0
- Else return the size of the file inside the option

Here's the code:

```
// helper to provide a default if missing
let ifNone deflt opt =
    defaultArg opt deflt

// get the file size of an option
let fileSize fileOpt =
    fileOpt
    |> Option.map (fun file -> file.fileSize)
    |> ifNone 0
```

Putting it all together then, we have our `largestFile` function:

```
let largestFile fileSystemItem =

    // helper to provide a default if missing
    let ifNone deflt opt =
        defaultArg opt deflt

    // helper to get the size of a File option
    let fileSize fileOpt =
        fileOpt
        |> Option.map (fun file -> file.fileSize)
        |> ifNone 0

    // handle File case
    let fFile (file:File) =
        Some file

    // handle Directory case
    let fDir (name,size,subfiles) =
        match subfiles with
        | [] ->
            None  // empty directory
        | subfiles ->
            // find the biggest File option using the helper
            subfiles
            |> List.maxBy fileSize

    // call the catamorphism
    cataFS fFile fDir fileSystemItem
```

If we test it, we get the results we expect:

```
readme |> largestFile
// Some {name = "readme.txt"; fileSize = 1}

src |> largestFile
// Some {name = "build.bat"; fileSize = 3}

bin |> largestFile
// None

root |> largestFile
// Some {name = "build.bat"; fileSize = 3}
```

Again, a little bit tricky to set up, but no more than if we had to write it from scratch without using a catamorphism at all.

Catamorphism example: Product domain

Let's work with a slightly more complicated domain. This time, imagine that we make and sell products of some kind:

- Some products are bought, with an optional vendor.
- Some products are made on our premises, built from subcomponents, where a subcomponent is some quantity of another product.

Here's the domain modelled as types:

```
type Product =
    | Bought of BoughtProduct
    | Made of MadeProduct
and BoughtProduct = {
    name : string
    weight : int
    vendor : string option }
and MadeProduct = {
    name : string
    weight : int
    components:Component list }
and Component = {
    qty : int
    product : Product }
```

Note that the types are mutally recursive. `Product` references `MadeProduct` which references `Component` which in turn references `Product` again.

Here are some example products:

```
let label =
    Bought {name="label"; weight=1; vendor=Some "ACME"}
let bottle =
    Bought {name="bottle"; weight=2; vendor=Some "ACME"}
let formulation =
    Bought {name="formulation"; weight=3; vendor=None}

let shampoo =
    Made {name="shampoo"; weight=10; components=
    [
    {qty=1; product=formulation}
    {qty=1; product=bottle}
    {qty=2; product=label}
    ]}

let twoPack =
    Made {name="twoPack"; weight=5; components=
    [
    {qty=2; product=shampoo}
    ]}
```

Now to design the catamorphism, we need to do is replace the `Product` type with `'r` in all the constructors.

Just as with the previous example, the `Bought` case is easy:

```
// case constructor
Bought  : BoughtProduct -> Product

// function parameter to handle Bought case
fBought : BoughtProduct -> 'r
```

The `Made` case is trickier. We need to expand the `MadeProduct` into a tuple. But that tuple contains a `Component`, so we need to expand that as well. Finally we get to the inner `Product`, and we can then mechanically replace that with `'r`.

Here's the sequence of transformations:

```
// case constructor
Made  : MadeProduct -> Product

// case constructor (MadeProduct unpacked as tuple)
Made  : (string,int,Component list) -> Product

// case constructor (Component unpacked as tuple)
Made  : (string,int,(int,Product) list) -> Product
//  replace with 'r ===> ~~~~~~~           ~~~~~~~

// function parameter to handle Made case
fMade : (string,int,(int,'r) list)      -> 'r
```

When implementing the `cataProduct` function we need to the same kind of mapping as before, turning a list of `Component` into a list of `(int,'r)`.

We'll need a helper for that:

```
// Converts a Component into a (int * 'r) tuple
let convertComponentToTuple comp =
    (comp.qty,recurse comp.product)
```

You can see that this uses the `recurse` function to turn the inner product (`comp.product`) into an `'r` and then make a tuple `int * 'r`.

With `convertComponentToTuple` available, we can convert all the components to tuples using `List.map` :

```
let componentTuples =
    made.components
    |> List.map convertComponentToTuple
```

`componentTuples` is a list of `(int * 'r)` , which is just what we need for the `fMade` function.

The complete implementation of `cataProduct` looks like this:

```
let rec cataProduct fBought fMade product :'r =
    let recurse = cataProduct fBought fMade

    // Converts a Component into a (int * 'r) tuple
    let convertComponentToTuple comp =
        (comp.qty,recurse comp.product)

    match product with
    | Bought bought ->
        fBought bought
    | Made made ->
        let componentTuples =  // (int * 'r) list
            made.components
            |> List.map convertComponentToTuple
        fMade (made.name,made.weight,componentTuples)
```

Product domain: `productWeight` example

We can now use `cataProduct` to calculate the weight, say.

```
let productWeight product =

    // handle Bought case
    let fBought (bought:BoughtProduct) =
        bought.weight

    // handle Made case
    let fMade (name,weight,componentTuples) =
        // helper to calculate weight of one component tuple
        let componentWeight (qty,weight) =
            qty * weight
        // add up the weights of all component tuples
        let totalComponentWeight =
            componentTuples
            |> List.sumBy componentWeight
        // and add the weight of the Made case too
        totalComponentWeight + weight

    // call the catamorphism
    cataProduct fBought fMade product
```

Let's test it interactively to make sure it works:

```
label |> productWeight    // 1
shampoo |> productWeight  // 17 = 10 + (2x1 + 1x2 + 1x3)
twoPack |> productWeight  // 39 = 5  + (2x17)
```

That's as we expect.

Try implementing `productWeight` from scratch, without using a helper function like `cataProduct`. Again, it's do-able, but you'll probably waste quite bit of time getting the recursion logic right.

Product domain: `mostUsedVendor` example

Let's do a more complex function now. What is the most used vendor?

The logic is simple: each time a product references a vendor, we'll give that vendor one point, and the vendor with the highest score wins.

Again, let's think about what it should return. That is, what is the `'r`?

You might think that it's just a score of some kind, but we also need to know the vendor name. Ok, a tuple then. But what if there are no vendors?

So let's make `'r` a `VendorScore option`, where we are going to create a little type `VendorScore`, rather than using a tuple.

```
type VendorScore = {vendor:string; score:int}
```

We'll also define some helpers to get data from a `VendorScore` easily:

```
let vendor vs = vs.vendor
let score vs = vs.score
```

Now, you can't determine the most used vendor over until you have results from the entire tree, so both the `Bought` case and the `Made` case need to return a list which can added to as we recurse up the tree. And then, after getting *all* the scores, we'll sort descending to find the vendor with the highest one.

So we have to make `'r` a `VendorScore list`, not just an option!

The logic for the `Bought` case is then:

- If the vendor is present, return a `VendorScore` with score = 1, but as a one-element list rather than as a single item.
- If the vendor is missing, return an empty list.

```
let fBought (bought:BoughtProduct) =
    // set score = 1 if there is a vendor
    bought.vendor
    |> Option.map (fun vendor -> {vendor = vendor; score = 1} )
    // => a VendorScore option
    |> Option.toList
    // => a VendorScore list
```

The function for the `Made` case is more complicated.

- If list of subscores is empty, then return an empty list.
- If list of subscores is non-empty, we sum them by vendor and then return the new list.

But the list of subresults passed into the `fMade` function will not be a list of subscores, it will be a list of tuples, `qty * 'r` where `'r` is `VendorScore list`. Complicated!

What we need to do then is:

- Turn `qty * 'r` into just `'r` because we don't care about the qty in this case. We now have a list of `VendorScore list`. We can use `List.map snd` to do this.
- But now we would have a list of `VendorScore list`. We can flatten a list of lists into a simple list using `List.collect`. And in fact, using `List.collect snd` can do both steps in one go.
- Group this list by vendor so that we have a list of `key=vendor; values=VendorScore list` tuples.
- Sum up the scores for each vendor (`values=VendorScore list`) into a single value, so that we have a list of `key=vendor; values=VendorScore` tuples.

At this point the `cata` function will return a `VendorScore list`. To get the highest score, use `List.sortByDescending` then `List.tryHead`. Note that `maxBy` won't work because the list could be empty.

Here's the complete `mostUsedVendor` function:

```
let mostUsedVendor product =

    let fBought (bought:BoughtProduct) =
        // set score = 1 if there is a vendor
        bought.vendor
        |> Option.map (fun vendor -> {vendor = vendor; score = 1} )
        // => a VendorScore option
        |> Option.toList
        // => a VendorScore list

    let fMade (name,weight,subresults) =
        // subresults are a list of (qty * VendorScore list)

        // helper to get sum of scores
        let totalScore (vendor,vendorScores) =
            let totalScore = vendorScores |> List.sumBy score
            {vendor=vendor; score=totalScore}

        subresults
        // => a list of (qty * VendorScore list)
        |> List.collect snd  // ignore qty part of subresult
        // => a list of VendorScore
        |> List.groupBy vendor
        // second item is list of VendorScore, reduce to sum
        |> List.map totalScore
        // => list of VendorScores

    // call the catamorphism
    cataProduct fBought fMade product
    |> List.sortByDescending score  // find highest score
    // return first, or None if list is empty
    |> List.tryHead
```

Now let's test it:

```
label |> mostUsedVendor
// Some {vendor = "ACME"; score = 1}

formulation |> mostUsedVendor
// None

shampoo |> mostUsedVendor
// Some {vendor = "ACME"; score = 2}

twoPack |> mostUsedVendor
// Some {vendor = "ACME"; score = 2}
```

This isn't the only possible implementation of `fMade` , of course. I could have used `List.fold` and done the whole thing in one pass, but this version seems like the most obvious and readable implementation.

It's also true that I could have avoided using `cataProduct` altogether and written `mostUsedVendor` from scratch. If performance is an issue, then that might be a better approach, because the generic catamorphism creates intermediate values (such as the list of `qty * VendorScore option`) which are over general and potentially wasteful.

On other hand, by using the catamorphism, I could focus on the counting logic only and ignore the recursion logic.

So as always, you should consider the pros and cons of reuse vs. creating from scratch; the benefits of writing common code once and using it in a standardized way, versus the performance but extra effort (and potential bugginess) of custom code.

Summary

We've seen in this post how to define a recursive type, and been introduced to catamorphisms.

And we have also seen some uses for catamorphisms:

- Any function that "collapses" a recursive type, such as `Gift -> 'r` , can be written in terms of the catamorphism for that type.
- Catamorphisms can be used to hide the internal structure of the type.
- Catamorphisms can be used to create mappings from one type to another by tweaking the functions that handle each case.
- Catamorphisms can be used to create a clone of the original value by passing in the type's case constructors.

But all is not perfect in the land of catamorphisms. In fact, all the catamorphism implementations on this page have a potentially serious flaw.

In the next post we'll see what can go wrong with them, how to fix them, and in the process look at the various kinds of "fold".

See you then!

The source code for this post is available at this gist.

UPDATE: Fixed logic error in `mostUsedVendor` *as pointed out by Paul Schnapp in comments. Thanks, Paul!*

Introducing Folds

This post is the third in a series.

In the first post, I introduced "catamorphisms", a way of creating functions for recursive types, and in the second post, we created a few catamorphism implementations.

But at the end of the previous post, I noted that all the catamorphism implementations so far have had a potentially serious flaw.

In this post, we'll look at the flaw and how to work around it, and in the process look at folds, tail-recursion and the difference between "left fold" and "right fold".

Series contents

Here's the contents of this series:

- **Part 1: Introduction to recursive types and catamorphisms**
  - A simple recursive type
  - Parameterize all the things
  - Introducing catamorphisms
  - Benefits of catamorphisms
  - Rules for creating a catamorphism
- **Part 2: Catamorphism examples**
  - Catamorphism example: File system domain
  - Catamorphism example: Product domain
- **Part 3: Introducing folds**
  - A flaw in our catamorphism implementation
  - Introducing `fold`
  - Problems with fold
  - Using functions as accumulators
  - Introducing `foldback`
  - Rules for creating a fold
- **Part 4: Understanding folds**
  - Iteration vs. recursion
  - Fold example: File system domain
  - Common questions about "fold"
- **Part 5: Generic recursive types**
  - LinkedList: A generic recursive type

- Making the Gift domain generic
- Defining a generic Container type
- A third way to implement the gift domain
- Abstract or concrete? Comparing the three designs

- **Part 6: Trees in the real world**
  - Defining a generic Tree type
  - The Tree type in the real world
  - Mapping the Tree type
  - Example: Creating a directory listing
  - Example: A parallel grep
  - Example: Storing the file system in a database
  - Example: Serializing a Tree to JSON
  - Example: Deserializing a Tree from JSON
  - Example: Deserializing a Tree from JSON - with error handling

---

A flaw in our catamorphism implementation

Before we look at the flaw, let's first review the recursive type `Gift` and the associated catamorphism `cataGift` that we created for it.

Here's the domain:

```
type Book = {title: string; price: decimal}

type ChocolateType = Dark | Milk | SeventyPercent
type Chocolate = {chocType: ChocolateType ; price: decimal}

type WrappingPaperStyle =
    | HappyBirthday
    | HappyHolidays
    | SolidColor

type Gift =
    | Book of Book
    | Chocolate of Chocolate
    | Wrapped of Gift * WrappingPaperStyle
    | Boxed of Gift
    | WithACard of Gift * message:string
```

Here are some example values that we'll be using in this post:

```
// A Book
let wolfHall = {title="Wolf Hall"; price=20m}
// A Chocolate
let yummyChoc = {chocType=SeventyPercent; price=5m}
// A Gift
let birthdayPresent = WithACard (Wrapped (Book wolfHall, HappyBirthday), "Happy Birthd
ay")
// A Gift
let christmasPresent = Wrapped (Boxed (Chocolate yummyChoc), HappyHolidays)
```

Here's the catamorphism:

```
let rec cataGift fBook fChocolate fWrapped fBox fCard gift :'r =
    let recurse = cataGift fBook fChocolate fWrapped fBox fCard
    match gift with
    | Book book ->
        fBook book
    | Chocolate choc ->
        fChocolate choc
    | Wrapped (gift,style) ->
        fWrapped (recurse gift,style)
    | Boxed gift ->
        fBox (recurse gift)
    | WithACard (gift,message) ->
        fCard (recurse gift,message)
```

and here is a `totalCostUsingCata` function built using `cataGift` :

```
let totalCostUsingCata gift =
    let fBook (book:Book) =
        book.price
    let fChocolate (choc:Chocolate) =
        choc.price
    let fWrapped  (innerCost,style) =
        innerCost + 0.5m
    let fBox innerCost =
        innerCost + 1.0m
    let fCard (innerCost,message) =
        innerCost + 2.0m
    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

What's the flaw?

So what is wrong with this implementation? Let's stress test it and find out!

What we'll do is create a `Box` inside a `Box` inside a `Box` a very large number of times, and see what happens.

Here's a little helper function to create nested boxes:

```
let deeplyNestedBox depth =
    let rec loop depth boxSoFar =
        match depth with
        | 0 -> boxSoFar
        | n -> loop (n-1) (Boxed boxSoFar)
    loop depth (Book wolfHall)
```

Let's try it to make sure it works:

```
deeplyNestedBox 5
// Boxed (Boxed (Boxed (Boxed (Boxed (Book {title = "Wolf Hall"; price = 20M})))))

deeplyNestedBox 10
//  Boxed(Boxed(Boxed(Boxed(Boxed
//   (Boxed(Boxed(Boxed(Boxed(Boxed(Book {title = "Wolf Hall";price = 20M}))))))))))
```

Now try running `totalCostUsingCata` with these deeply nested boxes:

```
deeplyNestedBox 10 |> totalCostUsingCata       // OK      30.0M
deeplyNestedBox 100 |> totalCostUsingCata      // OK     120.0M
deeplyNestedBox 1000 |> totalCostUsingCata     // OK    1020.0M
```

So far so good.

But if we use much larger numbers, we soon run into a stack overflow exception:

```
deeplyNestedBox 10000 |> totalCostUsingCata  // Stack overflow?
deeplyNestedBox 100000 |> totalCostUsingCata // Stack overflow?
```

The exact number which causes an error depends on the environment, available memory, and so on. But it is a certainty that you will run into it when you start using largish numbers.

Why is this happening?

The problem with deep recursion

Recall that the definition of the cost for the Boxed case (`fBox`) was `innerCost + 1.0m` . And what is the inner cost? That's another box too, so we end up with a chain of calculations looking like this:

```
innerCost + 1.0m where innerCost =
  innerCost2 + 1.0m where innerCost2 =
    innerCost3 + 1.0m where innerCost3 =
      innerCost4 + 1.0m where innerCost4 =
        ...
        innerCost999 + 1.0m where innerCost999 =
          innerCost1000 + 1.0m where innerCost1000 =
            book.price
```

In other words, `innerCost1000` has to be calculated before `innerCost999` can be calculated, and 999 other inner costs have to be calculated before the top level `innerCost` can be calculated.

Every level is waiting for its inner cost to be calculated before doing the calculation for that level.

All these unfinished calculations are stacked up waiting for the inner one to complete. And when you have too many? Boom! Stack overflow!

The solution to stack overflows

The solution to this problem is simple. Rather than each level waiting for the inner cost to be calculated, each level calculates the cost so far, using an accumulator, and passes that down to the next inner level. When we get to the bottom level, we have the final answer.

```
costSoFar = 1.0m; evaluate calcInnerCost with costSoFar:
  costSoFar = costSoFar + 1.0m; evaluate calcInnerCost with costSoFar:
    costSoFar = costSoFar + 1.0m; evaluate calcInnerCost with costSoFar:
      costSoFar = costSoFar + 1.0m; evaluate calcInnerCost with costSoFar:
        ...
        costSoFar = costSoFar + 1.0m; evaluate calcInnerCost with costSoFar:
          costSoFar = costSoFar + 1.0m; evaluate calcInnerCost with costSoFar:
            finalCost = costSoFar + book.price   // final answer
```

The big advantange of this approach is that all calculations at a particular level are *completely finished* before the next lowel level is called. Which means that the level and its associated data can be safely discarded from the stack. Which means no stack overflow!

An implementation like this, where the higher levels can be safely discarded, is called *tail recursive*.

Reimplementating the `totalCost` function with an accumulator

Let's rewrite the total cost function from scratch, using an accumulator called `costSoFar` :

```
let rec totalCostUsingAcc costSoFar gift =
    match gift with
    | Book book ->
        costSoFar + book.price  // final result
    | Chocolate choc ->
        costSoFar + choc.price  // final result
    | Wrapped (innerGift,style) ->
        let newCostSoFar = costSoFar + 0.5m
        totalCostUsingAcc newCostSoFar innerGift
    | Boxed innerGift ->
        let newCostSoFar = costSoFar + 1.0m
        totalCostUsingAcc newCostSoFar innerGift
    | WithACard (innerGift,message) ->
        let newCostSoFar = costSoFar + 2.0m
        totalCostUsingAcc newCostSoFar innerGift
```

A few things to note:

- The new version of the function has an extra parameter (`costSoFar`). We will have to provide an initial value for this (such as zero) when we call it at the top level.
- The non-recursive cases (`Book` and `Chocolate`) are the end points. The take the cost so far and add it to their price, and then that is the final result.
- The recursive cases calculate a new `costSoFar` based on the parameter that is passed in. The new `costSoFar` is then passed down to the next lower level, just as in the example above.

Let's stress test this version:

```
deeplyNestedBox 1000 |> totalCostUsingAcc 0.0m     // OK    1020.0M
deeplyNestedBox 10000 |> totalCostUsingAcc 0.0m    // OK   10020.0M
deeplyNestedBox 100000 |> totalCostUsingAcc 0.0m   // OK  100020.0M
deeplyNestedBox 1000000 |> totalCostUsingAcc 0.0m  // OK 1000020.0M
```

Excellent. Up to one million nested levels without a hiccup.

Introducing "fold"

Now let's apply the same design principle to the catamorphism implementation.

We'll create a new function `foldGift` . We'll introduce an accumulator `acc` that we will thread through each level, and the non-recursive cases will return the final accumulator.

```
let rec foldGift fBook fChocolate fWrapped fBox fCard acc gift :'r =
    let recurse = foldGift fBook fChocolate fWrapped fBox fCard
    match gift with
    | Book book ->
        let finalAcc = fBook acc book
        finalAcc     // final result
    | Chocolate choc ->
        let finalAcc = fChocolate acc choc
        finalAcc     // final result
    | Wrapped (innerGift,style) ->
        let newAcc = fWrapped acc style
        recurse newAcc innerGift
    | Boxed innerGift ->
        let newAcc = fBox acc
        recurse newAcc innerGift
    | WithACard (innerGift,message) ->
        let newAcc = fCard acc message
        recurse newAcc innerGift
```

If we look at the type signature, we can see that it is subtly different. The type of the accumulator `'a` is being used everywhere now. The only time where the final return type is used is in the two non-recursive cases (`fBook` and `fChocolate`).

```
val foldGift :
  fBook:('a -> Book -> 'r) ->
  fChocolate:('a -> Chocolate -> 'r) ->
  fWrapped:('a -> WrappingPaperStyle -> 'a) ->
  fBox:('a -> 'a) ->
  fCard:('a -> string -> 'a) ->
  // accumulator
  acc:'a ->
  // input value
  gift:Gift ->
  // return value
  'r
```

Let's look at this more closely, and compare the signatures of the original catamorphism from the last post with the signatures of the new `fold` function.

First of all, the non-recursive cases:

```
// original catamorphism
fBook:(Book -> 'r)
fChocolate:(Chocolate -> 'r)

// fold
fBook:('a -> Book -> 'r)
fChocolate:('a -> Chocolate -> 'r)
```

As you can see, with "fold", the non-recursive cases take an extra parameter (the accumulator) and return the `'r` type.

This is a very important point: *the type of the accumulator does not need to be the same as the return type.* We will need to take advantage of this shortly.

What about the recursive cases? How did their signature change?

```
// original catamorphism
fWrapped:('r -> WrappingPaperStyle -> 'r)
fBox:('r -> 'r)

// fold
fWrapped:('a -> WrappingPaperStyle -> 'a)
fBox:('a -> 'a)
```

For the recursive cases, the structure is identical but all use of the `'r` type has been replaced with the `'a` type. The recursive cases do not use the `'r` type at all.

Reimplementating the `totalCost` function using fold

Once again, we can reimplement the total cost function, but this time using the `foldGift` function:

```
let totalCostUsingFold gift =

    let fBook costSoFar (book:Book) =
        costSoFar + book.price
    let fChocolate costSoFar (choc:Chocolate) =
        costSoFar + choc.price
    let fWrapped costSoFar style =
        costSoFar + 0.5m
    let fBox costSoFar =
        costSoFar + 1.0m
    let fCard costSoFar message =
        costSoFar + 2.0m

    // initial accumulator
    let initialAcc = 0m

    // call the fold
    foldGift fBook fChocolate fWrapped fBox fCard initialAcc gift
```

And again, we can process very large numbers of nested boxes without a stack overflow:

```
deeplyNestedBox 100000 |> totalCostUsingFold  // no problem   100020.0M
deeplyNestedBox 1000000 |> totalCostUsingFold // no problem  1000020.0M
```

Problems with fold

So using fold solves all our problems, right?

Unfortunately, no.

Yes, there are no more stack overflows, but we have another problem now.

Reimplementation of the `description` function

To see what the problem is, let's revisit the `description` function that we created in the first post.

The original one was not tail-recursive, so let's make it safer and reimplement it using `foldGift`.

```
let descriptionUsingFold gift =
    let fBook descriptionSoFar (book:Book) =
        sprintf "'%s' %s" book.title descriptionSoFar

    let fChocolate descriptionSoFar (choc:Chocolate) =
        sprintf "%A chocolate %s" choc.chocType descriptionSoFar

    let fWrapped descriptionSoFar style =
        sprintf "%s wrapped in %A paper" descriptionSoFar style

    let fBox descriptionSoFar =
        sprintf "%s in a box" descriptionSoFar

    let fCard descriptionSoFar message =
        sprintf "%s with a card saying '%s'" descriptionSoFar message

    // initial accumulator
    let initialAcc = ""

    // main call
    foldGift fBook fChocolate fWrapped fBox fCard initialAcc gift
```

Let's see what the output is:

```
birthdayPresent |> descriptionUsingFold
// "'Wolf Hall'  with a card saying 'Happy Birthday' wrapped in HappyBirthday paper"

christmasPresent |> descriptionUsingFold
// "SeventyPercent chocolate  wrapped in HappyHolidays paper in a box"
```

These outputs are wrong! The order of the decorations has been mixed up.

It's supposed to be a wrapped book with a card, not a book and a card wrapped together. And it's supposed to be chocolate in a box, then wrapped, not wrapped chocolate in a box!

```
// OUTPUT: "'Wolf Hall'  with a card saying 'Happy Birthday' wrapped in HappyBirthday
paper"
// CORRECT "'Wolf Hall' wrapped in HappyBirthday paper with a card saying 'Happy Birth
day'"

// OUTPUT: "SeventyPercent chocolate  wrapped in HappyHolidays paper in a box"
// CORRECT "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"
```

What has gone wrong?

The answer is that the correct description for each layer depends on the description of the layer below. We can't "pre-calculate" the description for a layer and pass it down to the next layer using a `descriptionSoFar` accumulator.

But now we have a dilemma: a layer depends on information from the layer below, but we want to avoid a stack overflow.

Using functions as accumulators

Remember that the accumulator type does not have to be the same as the return type. We can use anything as an accumulator, even a function!

So what we'll do is, rather than passing a `descriptionSoFar` as the accumulator, we'll pass a function (`descriptionGenerator` say) that will build the appropriate description given the value of the next layer down.

Here's the implementation for the non-recursive cases:

```
let fBook descriptionGenerator (book:Book) =
    descriptionGenerator (sprintf "'%s'" book.title)
//  ~~~~~~~~~~~~~~~~~~~~  <= a function as an accumulator!

let fChocolate descriptionGenerator (choc:Chocolate) =
    descriptionGenerator (sprintf "%A chocolate" choc.chocType)
```

The implementation for recursive cases is a bit more complicated:

- We are given an accumulator (`descriptionGenerator`) as a parameter.
- We need to create a new accumulator (a new `descriptionGenerator`) to pass down to the next lower layer.
- The *input* to the description generator will be all the data accumulated from the lower layers. We manipulate that to make a new description and then call the `descriptionGenerator` passed in from the higher layer.

It's more complicated to talk about than to demonstrate, so here are implementations for two of the cases:

```
let fWrapped descriptionGenerator style =
    let newDescriptionGenerator innerText =
        let newInnerText = sprintf "%s wrapped in %A paper" innerText style
        descriptionGenerator newInnerText
    newDescriptionGenerator

let fBox descriptionGenerator =
    let newDescriptionGenerator innerText =
        let newInnerText = sprintf "%s in a box" innerText
        descriptionGenerator newInnerText
    newDescriptionGenerator
```

We can simplify that code a little by using a lambda directly:

```
let fWrapped descriptionGenerator style =
    fun innerText ->
        let newInnerText = sprintf "%s wrapped in %A paper" innerText style
        descriptionGenerator newInnerText

let fBox descriptionGenerator =
    fun innerText ->
        let newInnerText = sprintf "%s in a box" innerText
        descriptionGenerator newInnerText
```

We could continue to make it more compact using piping and other things, but I think that what we have here is a good balance between conciseness and obscurity.

Here is the entire function:

```
let descriptionUsingFoldWithGenerator gift =

    let fBook descriptionGenerator (book:Book) =
        descriptionGenerator (sprintf "'%s'" book.title)

    let fChocolate descriptionGenerator (choc:Chocolate) =
        descriptionGenerator (sprintf "%A chocolate" choc.chocType)

    let fWrapped descriptionGenerator style =
        fun innerText ->
            let newInnerText = sprintf "%s wrapped in %A paper" innerText style
            descriptionGenerator newInnerText

    let fBox descriptionGenerator =
        fun innerText ->
            let newInnerText = sprintf "%s in a box" innerText
            descriptionGenerator newInnerText

    let fCard descriptionGenerator message =
        fun innerText ->
            let newInnerText = sprintf "%s with a card saying '%s'" innerText message
            descriptionGenerator newInnerText

    // initial DescriptionGenerator
    let initialAcc = fun innerText -> innerText

    // main call
    foldGift fBook fChocolate fWrapped fBox fCard initialAcc gift
```

Again, I'm using overly descriptive intermediate values to make it clear what is going on.

If we try `descriptionUsingFoldWithGenerator` now, we get the correct answers again:

```
birthdayPresent |> descriptionUsingFoldWithGenerator
// CORRECT "'Wolf Hall' wrapped in HappyBirthday paper with a card saying 'Happy Birth
day'"

christmasPresent |> descriptionUsingFoldWithGenerator
// CORRECT "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"
```

Introducing "foldback"

Now that we understand what to do, let's make a generic version that that handles the generator function logic for us. This one we will call "foldback":

By the way, I'm going to use term "generator" here. In other places, it is commonly referred to as a "continuation" function, often abbreviated to just "k".

Here's the implementation:

```
let rec foldbackGift fBook fChocolate fWrapped fBox fCard generator gift :'r =
    let recurse = foldbackGift fBook fChocolate fWrapped fBox fCard
    match gift with
    | Book book ->
        generator (fBook book)
    | Chocolate choc ->
        generator (fChocolate choc)
    | Wrapped (innerGift,style) ->
        let newGenerator innerVal =
            let newInnerVal = fWrapped innerVal style
            generator newInnerVal
        recurse newGenerator innerGift
    | Boxed innerGift ->
        let newGenerator innerVal =
            let newInnerVal = fBox innerVal
            generator newInnerVal
        recurse newGenerator innerGift
    | WithACard (innerGift,message) ->
        let newGenerator innerVal =
            let newInnerVal = fCard innerVal message
            generator newInnerVal
        recurse newGenerator innerGift
```

You can see that it is just like the `descriptionUsingFoldWithGenerator` implementation, except that we are using generic `newInnerVal` and `generator` values.

The type signatures are similar to the original catamorphism, except that every case works with `'a` only now. The only time `'r` is used is in the generator function itself!

```
val foldbackGift :
  fBook:(Book -> 'a) ->
  fChocolate:(Chocolate -> 'a) ->
  fWrapped:('a -> WrappingPaperStyle -> 'a) ->
  fBox:('a -> 'a) ->
  fCard:('a -> string -> 'a) ->
  // accumulator
  generator:('a -> 'r) ->
  // input value
  gift:Gift ->
  // return value
  'r
```

The `foldback` implementation above is written from scratch. If you want a fun exercise, see if you can write `foldback` in terms of `fold`.

Let's rewrite the `description` function using `foldback` :

```
let descriptionUsingFoldBack gift =
    let fBook (book:Book) =
        sprintf "'%s'" book.title
    let fChocolate (choc:Chocolate) =
        sprintf "%A chocolate" choc.chocType
    let fWrapped innerText style =
        sprintf "%s wrapped in %A paper" innerText style
    let fBox innerText =
        sprintf "%s in a box" innerText
    let fCard innerText message =
        sprintf "%s with a card saying '%s'" innerText message
    // initial DescriptionGenerator
    let initialAcc = fun innerText -> innerText
    // main call
    foldbackGift fBook fChocolate fWrapped fBox fCard initialAcc gift
```

And the results are still correct:

```
birthdayPresent |> descriptionUsingFoldBack
// CORRECT "'Wolf Hall' wrapped in HappyBirthday paper with a card saying 'Happy Birth
day'"

christmasPresent |> descriptionUsingFoldBack
// CORRECT "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"
```

Comparing `foldback` to the original catamorphism

The implementation of `descriptionUsingFoldBack` is almost identical to the version in the last post that used the original catamorphism `cataGift` .

Here's the version using `cataGift` :

```
let descriptionUsingCata gift =
    let fBook (book:Book) =
        sprintf "'%s'" book.title
    let fChocolate (choc:Chocolate) =
        sprintf "%A chocolate" choc.chocType
    let fWrapped (innerText,style) =
        sprintf "%s wrapped in %A paper" innerText style
    let fBox innerText =
        sprintf "%s in a box" innerText
    let fCard (innerText,message) =
        sprintf "%s with a card saying '%s'" innerText message
    // call the catamorphism
    cataGift fBook fChocolate fWrapped fBox fCard gift
```

And here's the version using `foldbackGift`:

```
let descriptionUsingFoldBack gift =
    let fBook (book:Book) =
        sprintf "'%s'" book.title
    let fChocolate (choc:Chocolate) =
        sprintf "%A chocolate" choc.chocType
    let fWrapped innerText style =
        sprintf "%s wrapped in %A paper" innerText style
    let fBox innerText =
        sprintf "%s in a box" innerText
    let fCard innerText message =
        sprintf "%s with a card saying '%s'" innerText message
    // initial DescriptionGenerator
    let initialAcc = fun innerText -> innerText    // could be replaced with id
    // main call
    foldbackGift fBook fChocolate fWrapped fBox fCard initialAcc gift
```

All the handler functions are basically identical. The only change is the addition of an initial generator function, which is just `id` in this case.

However, although the code looks the same in both cases, they differ in their recursion safety. The `foldbackGift` version is still tail recursive, and can handle very large nesting depths, unlike the `cataGift` version.

But this implementation is not perfect either. The chain of nested functions can get very slow and generate a lot of garbage, and for this particular example, there is an even faster way, which we'll look at in the next post.

Changing the type signature of `foldback` to avoid a mixup

In `foldGift` the signature for `fWrapped` is:

```
fWrapped:('a -> WrappingPaperStyle -> 'a)
```

But in `foldbackGift` the signature for `fWrapped` is:

```
fWrapped:('a -> WrappingPaperStyle -> 'a)
```

Can you spot the difference? No, me neither.

The two functions are very similar, yet work very differently. In the `foldGift` version, the first parameter is the accumulator from the *outer* levels, while in `foldbackGift` version, the first parameter is the accumulator from the *inner* levels. Quite an important distinction!

It is therefore common to change the signature of the `foldBack` version so that the accumulator always comes *last*, while in the normal `fold` function, the accumulator always comes *first*.

```
let rec foldbackGift fBook fChocolate fWrapped fBox fCard gift generator :'r =
//swapped =>                                             ~~~~~~~~~~~~~~

    let recurse = foldbackGiftWithAccLast fBook fChocolate fWrapped fBox fCard

    match gift with
    | Book book ->
        generator (fBook book)
    | Chocolate choc ->
        generator (fChocolate choc)

    | Wrapped (innerGift,style) ->
        let newGenerator innerVal =
            let newInnerVal = fWrapped style innerVal
//swapped =>                              ~~~~~~~~~~~~~~
            generator newInnerVal
        recurse innerGift newGenerator
//swapped =>    ~~~~~~~~~~~~~~~~~~~~~~

    | Boxed innerGift ->
        let newGenerator innerVal =
            let newInnerVal = fBox innerVal
            generator newInnerVal
        recurse innerGift newGenerator
//swapped =>    ~~~~~~~~~~~~~~~~~~~~~~

    | WithACard (innerGift,message) ->
        let newGenerator innerVal =
            let newInnerVal = fCard message innerVal
//swapped =>                           ~~~~~~~~~~~~~~~~
            generator newInnerVal
        recurse innerGift newGenerator
//swapped =>    ~~~~~~~~~~~~~~~~~~~~~~
```

This change shows up in the type signature. The `Gift` value comes before the accumulator now:

```
val foldbackGift :
  fBook:(Book -> 'a) ->
  fChocolate:(Chocolate -> 'a) ->
  fWrapped:(WrappingPaperStyle -> 'a -> 'a) ->
  fBox:('a -> 'a) ->
  fCard:(string -> 'a -> 'a) ->
  // input value
  gift:Gift ->
  // accumulator
  generator:('a -> 'r) ->
  // return value
  'r
```

and now we *can* tell the two versions apart easily.

```
// fold
fWrapped:('a -> WrappingPaperStyle -> 'a)

// foldback
fWrapped:(WrappingPaperStyle -> 'a -> 'a)
```

Rules for creating a fold

To finish up this post, let's summarize the rules for creating a fold.

In the first post we saw that creating a catamorphism is a mechanical process that follows rules. The same is true for creating a iterative top-down fold. The process is:

- Create a function parameter to handle each case in the structure.
- Add an additional parameter as an accumulator.
- For non-recursive cases, pass the function parameter the accumulator plus all the data associated with that case.
- For recursive cases, perform two steps:
  - First, pass the handler the accumulator plus all the data associated with that case (except the inner recursive data). The result is a new accumulator value.
  - Then, call the fold recursively on the nested value using the new accumulator value.

Note that each handler only "sees" (a) the data for that case, and (b) the accumulator passed to it from the outer level. It does not have access to the results from the inner levels.

Summary

We've seen in this post how to define a tail-recursive implementation of a catamorphism, called "fold" and the reverse version "foldback".

In the next post we'll step back a bit and spend some time understanding what "fold" really means, and at some guidelines for choosing between `fold`, `foldback` and `cata`.

We'll then see if we can apply these rules to another domain.

The source code for this post is available at this gist.

Understanding Folds

This post is the fourth in a series.

In the previous post, I introduced "folds", a way of creating top-down iterative functions for recursive types.

In this post, we'll spend some time understanding folds in more detail.

Series contents

Here's the contents of this series:

- **Part 1: Introduction to recursive types and catamorphisms**
  - A simple recursive type
  - Parameterize all the things
  - Introducing catamorphisms
  - Benefits of catamorphisms
  - Rules for creating a catamorphism
- **Part 2: Catamorphism examples**
  - Catamorphism example: File system domain
  - Catamorphism example: Product domain
- **Part 3: Introducing folds**
  - A flaw in our catamorphism implementation
  - Introducing `fold`
  - Problems with fold
  - Using functions as accumulators
  - Introducing `foldback`
  - Rules for creating a fold
- **Part 4: Understanding folds**
  - Iteration vs. recursion
  - Fold example: File system domain
  - Common questions about "fold"
- **Part 5: Generic recursive types**
  - LinkedList: A generic recursive type
  - Making the Gift domain generic
  - Defining a generic Container type
  - A third way to implement the gift domain
  - Abstract or concrete? Comparing the three designs

- **Part 6: Trees in the real world**
  - Defining a generic Tree type
  - The Tree type in the real world
  - Mapping the Tree type
  - Example: Creating a directory listing
  - Example: A parallel grep
  - Example: Storing the file system in a database
  - Example: Serializing a Tree to JSON
  - Example: Deserializing a Tree from JSON
  - Example: Deserializing a Tree from JSON - with error handling

---

Iteration vs. recursion

We now have *three* different functions -- `cata` , `fold` and `foldback` .

So what exactly are the differences between them? We've seen that something that doesn't work with `fold` will work with `foldBack` , but is there an easy way to remember the difference?

One way to differentiate the three approaches is by remembering this:

- `fold` is top-down *iteration*
- `cata` is bottom-up *recursion*
- `foldBack` is bottom-up *iteration*

What does this mean?

Well, for in `fold` , the accumulator was initialized at the top level, and was passed down to each lower level until the lowest and last level was reached.

In code terms, each level did this:

```
accumulatorFromHigherLevel, combined with
  stuffFromThisLevel
    => stuffToSendDownToNextLowerLevel
```

In an imperative language, this is exactly a "for loop" with a mutable variable storing the accumulator.

```
var accumulator = initialValue
foreach level in levels do
{
  accumulator, combined with
    stuffFromThisLevel
      => update accumulator
}
```

So, this kind of top-to-bottom folding can be thought of as iteration (and in fact, the F# compiler will turn a tail-recursive function like this into an iteration behind the scenes).

On the other hand, in `cata`, the accumulator started at the bottom level, and was passed up to each higher level until the top level was reached.

In code terms, each level did this:

```
accumulatorFromLowerLevel, combined with
  stuffFromThisLevel
    => stuffToSendUpToNextHigherLevel
```

This is exactly a recursive loop:

```
let recurse (head::tail) =
    if atBottomLevel then
       return something
    else    // if not at bottom level
       let accumulatorFromLowerLevel = recurse tail
       return stuffFromThisLevel, combined with
          accumulatorFromLowerLevel
```

Finally, `foldback` can be thought of as "reverse iteration". The accumulator is threaded through all the levels, but starting at the bottom rather than at the top. It has the benefits of `cata` in that the inner values are calculated first and passed back up, but because it is iterative, there cannot be a stack overflow.

Many of the concepts we have discussed so far become clear when expressed in terms of iteration vs. recursion. For example:

- The iterative versions (`fold` and `foldback`) have no stack, and cannot cause a stack overflow.
- The "total cost" function needed no inner data, and so the top-down iterative version (`fold`) worked without problems.
- The "description" function though, needed inner text for correct formatting, and so the recursive version (`cata`) or bottom up iteration (`foldback`) was more suitable.

Fold example: File system domain

In the last post, we described some rules for creating folds. Let's see if we can apply these rules to create a fold in another domain, the "File System" domain from the second post in the series.

As a reminder, here is the crude "file system" domain from that post:

```
type FileSystemItem =
    | File of File
    | Directory of Directory
and File = {name:string; fileSize:int}
and Directory = {name:string; dirSize:int; subitems:FileSystemItem list}
```

Note that each directory contains a *list* of subitems, so this is not a linear structure like `Gift`, but a tree-like structure. Out implementation of fold will have to take this into account.

Here are some sample values:

```
let readme = File {name="readme.txt"; fileSize=1}
let config = File {name="config.xml"; fileSize=2}
let build  = File {name="build.bat"; fileSize=3}
let src = Directory {name="src"; dirSize=10; subitems=[readme; config; build]}
let bin = Directory {name="bin"; dirSize=10; subitems=[]}
let root = Directory {name="root"; dirSize=5; subitems=[src; bin]}
```

We want to create a fold, `foldFS`, say. So, following the rules, let's add an extra accumulator parameter `acc` and pass it to the `File` case:

```
let rec foldFS fFile fDir acc item :'r =
    let recurse = foldFS fFile fDir
    match item with
    | File file ->
        fFile acc file
    | Directory dir ->
        // to do
```

The `Directory` case is trickier. We are not supposed to know about the subitems, so that means that the only data we can use is the `name`, `dirSize`, and the accumulator passed in from a higher level. These are combined to make a new accumulator.

```
| Directory dir ->
    let newAcc = fDir acc (dir.name,dir.dirSize)
    // to do
```

NOTE: I'm keeping the `name` and `dirSize` as a tuple for grouping purposes, but of course you could pass them in as separate parameters.

Now we need to pass this new accumulator down to each subitem in turn, but each subitem will return a new accumulator of its own, so we need to use the following approach:

- Take the newly created accumulator and pass it to the first subitem.
- Take the output of that (another accumulator) and pass it to the second subitem.
- Take the output of that (another accumulator) and pass it to the third subitem.
- And so on. The output of the last subitem is the final result.

That approach is already available to us though. It's exactly what `List.fold` does! So here's the code for the Directory case:

```
| Directory dir ->
    let newAcc = fDir acc (dir.name,dir.dirSize)
    dir.subitems |> List.fold recurse newAcc
```

And here's the entire `foldFS` function:

```
let rec foldFS fFile fDir acc item :'r =
    let recurse = foldFS fFile fDir
    match item with
    | File file ->
        fFile acc file
    | Directory dir ->
        let newAcc = fDir acc (dir.name,dir.dirSize)
        dir.subitems |> List.fold recurse newAcc
```

With this in place, we can rewrite the same two functions we implemented in the last post.

First, the `totalSize` function, which just sums up all the sizes:

```
let totalSize fileSystemItem =
    let fFile acc (file:File) =
        acc + file.fileSize
    let fDir acc (name,size) =
        acc + size
    foldFS fFile fDir 0 fileSystemItem
```

And if we test it we get the same results as before:

```
readme |> totalSize  // 1
src |> totalSize     // 16 = 10 + (1 + 2 + 3)
root |> totalSize    // 31 = 5 + 16 + 10
```

File system domain: `largestFile` example

We can also reimplement the "what is the largest file in the tree?" function.

As before it will return a `File option`, because the tree might be empty. This means that the accumulator will be a `File option` too.

This time it is the `File` case handler which is tricky:

- If the accumulator being passed in is `None`, then this current file becomes the new accumulator.
- If the accumulator being passed in is `Some file`, then compare the size of that file with this file. Whichever is bigger becomes the new accumulator.

Here's the code:

```
let fFile (largestSoFarOpt:File option) (file:File) =
    match largestSoFarOpt with
    | None ->
        Some file
    | Some largestSoFar ->
        if largestSoFar.fileSize > file.fileSize then
            Some largestSoFar
        else
            Some file
```

On the other hand, the `Directory` handler is trivial -- just pass the "largest so far" accumulator down to the next level

```
let fDir largestSoFarOpt (name,size) =
    largestSoFarOpt
```

Here's the complete implementation:

```
let largestFile fileSystemItem =
    let fFile (largestSoFarOpt:File option) (file:File) =
        match largestSoFarOpt with
        | None ->
            Some file
        | Some largestSoFar ->
            if largestSoFar.fileSize > file.fileSize then
                Some largestSoFar
            else
                Some file

    let fDir largestSoFarOpt (name,size) =
        largestSoFarOpt

    // call the fold
    foldFS fFile fDir None fileSystemItem
```

And if we test it we get the same results as before:

```
readme |> largestFile
// Some {name = "readme.txt"; fileSize = 1}

src |> largestFile
// Some {name = "build.bat"; fileSize = 3}

bin |> largestFile
// None

root |> largestFile
// Some {name = "build.bat"; fileSize = 3}
```

It is interesting to compare this implementation with the recursive version in the second post. I think that this one is easier to implement, myself.

Tree traversal types

The various fold functions discussed so far correspond to various kinds of tree traversals:

- A `fold` function (as implemented here) is more properly called a "pre-order depth-first" tree traversal.
- A `foldback` function would be a "post-order depth-first" tree traversal.
- A `cata` function would not be a "traversal" at all, because each internal node deals with a list of all the subresults at once.

By tweaking the logic, you can make other variants.

For a description of the various kinds of tree traversals, see Wikipedia.

Do we need `foldback` ?

Do we need to implement a `foldback` function for the FileSystem domain?

I don't think so. If we need access to the inner data, we can just use the original "naive" catamorphism implementation in the previous post.

But, hey wait, didn't I say at the beginning that we had to watch out for stack overflows?

Yes, if the recursive type is deeply nested. But consider a file system with only two subdirectories per directory. How many directories would there be if there were 64 nested levels? (Hint: you may be familiar with a similar problem. Something to do with grains on a chessboard).

We saw earlier that the stack overflow issue only occurs with more than 1000 nested levels, and that level of nesting generally only occurs with *linear* recursive types, not trees like the FileSystem domain.

Common questions about "fold"

At this point you might be getting overwhelmed! All these different implementations with different advantages and disadvantages.

So let's take a short break and address some common questions.

What's the difference between "left fold" and "right fold"

There is often quite a lot of confusion around the terminology of folds: "left" vs. "right", "forward" vs. "backwards", etc.

- A *left fold* or *forward fold* is what I have just called `fold` -- the top-down iterative approach.
- A *right fold* or *backward fold* is what I have called `foldBack` -- the bottom-up iterative approach.

These terms, though, really only apply to linear recursive structures like `Gift`. When it comes to more complex tree-like structures, these distinctions are too simple, because there are many ways to traverse them: breadth-first, depth-first, pre-order and post-order, and so on.

Which type of fold function should I use?

Here are some guidelines:

- If your recursive type is not going to be too deeply nested (less than 100 levels deep, say), then the naive `cata` catamorphism we described in the first post is fine. It's really easy to implement mechanically -- just replace the main recursive type with `'r`.
- If your recursive type is going to be deeply nested and you want to prevent stack overflows, use the iterative `fold`.
- If you are using an iterative fold but you need to have access to the inner data, pass a continuation function as an accumulator.
- Finally, the iterative approach is generally faster and uses less memory than the recursive approach (but that advantage is lost if you pass around too many nested continuations).

Another way to think about it is to look at your "combiner" function. At each step, you are combining data from the different levels:

```
level1 data [combined with] level2 data [combined with] level3 data [combined with] le
vel4 data
```

If your combiner function is "left associative" like this:

```
(((level1 combineWith level2) combineWith level3) combineWith level4)
```

then use the iterative approach, but if your combiner function is "right associative" like this:

```
(level1 combineWith (level2 combineWith (level3 combineWith level4)))
```

then use the `cata` or `foldback` approach.

And if your combiner function doesn't care (like addition, for example), use whichever one is more convenient.

How can I know whether code is tail-recursive or not?

It's not always obvious whether an implementation is tail-recursive or not. The easiest way to be sure is to look at the very last expression for each case.

If the call to "recurse" is the very last expression, then it is tail-recursive. If there is any other work after that, then it is not tail-recursive.

See for yourself with the three implementations that we have discussed.

First, here's the code for the `WithACard` case in the original `cataGift` implementation:

```
| WithACard (gift,message) ->
    fCard (recurse gift,message)
//          ~~~~~~~  <= Call to recurse is not last expression.
//                      Tail-recursive? No!
```

The `cataGift` implementation is *not* tail-recursive.

Here's the code from the `foldGift` implementation:

```
| WithACard (innerGift,message) ->
    let newAcc = fCard acc message
    recurse newAcc innerGift
//  ~~~~~~~  <= Call to recurse is last expression.
//              Tail-recursive? Yes!
```

The `foldGift` implementation *is* tail-recursive.

And here's the code from the `foldbackGift` implementation:

```
| WithACard (innerGift,message) ->
    let newGenerator innerVal =
        let newInnerVal = fCard innerVal message
        generator newInnerVal
    recurse newGenerator innerGift
//  ~~~~~~~  <= Call to recurse is last expression.
//              Tail-recursive? Yes!
```

The `foldbackGift` implementation is also tail-recursive.

How do I short-circuit a fold?

In a language like C#, you can exit a iterative loop early using `break` like this:

```
foreach (var elem in collection)
{
    // do something

    if ( x == "error")
    {
        break;
    }
}
```

So how do you do the same thing with a fold?

The short answer is, you can't! A fold is designed to visit all elements in turn. The Visitor Pattern has the same constraint.

There are three workarounds.

The first one is to not use `fold` at all and create your own recursive function that terminates on the required condition:

In this example, the loop exits when the sum is larger than 100:

```
let rec firstSumBiggerThan100 sumSoFar listOfInts =
    match listOfInts with
    | [] ->
        sumSoFar // exhausted all the ints!
    | head::tail ->
        let newSumSoFar = head + sumSoFar
        if newSumSoFar > 100 then
            newSumSoFar
        else
            firstSumBiggerThan100 newSumSoFar tail

// test
[30;40;50;60] |> firstSumBiggerThan100 0  // 120
[1..3..100] |> firstSumBiggerThan100 0  // 117
```

The second approach is to use `fold` but to add some kind of "ignore" flag to the accumulator that is passed around. Once this flag is set, the remaining iterations do nothing.

Here's an example of calculating the sum, but the accumulator is actually a tuple with an `ignoreFlag` in addition to the `sumSoFar`:

```
let firstSumBiggerThan100 listOfInts =

    let folder accumulator i =
        let (ignoreFlag,sumSoFar) = accumulator
        if not ignoreFlag then
            let newSumSoFar = i + sumSoFar
            let newIgnoreFlag  = newSumSoFar > 100
            (newIgnoreFlag, newSumSoFar)
        else
            // pass the accumulator along
            accumulator

    let initialAcc = (false,0)

    listOfInts
    |> List.fold folder initialAcc  // use fold
    |> snd // get the sumSoFar

/// test
[30;40;50;60] |> firstSumBiggerThan100  // 120
[1..3..100] |> firstSumBiggerThan100  // 117
```

The third version is a variant of the second -- create a special value to signal that the remaining data should be ignored, but wrap it in a computation expression so that it looks more natural.

This approach is documented on Tomas Petricek's blog and the code looks like this:

```
let firstSumBiggerThan100 listOfInts =
    let mutable sumSoFar = 0
    imperative {
        for x in listOfInts do
            sumSoFar <- x + sumSoFar
            if sumSoFar > 100 then do! break
    }
    sumSoFar
```

Summary

The goal of this post was to help you understand folds better, and to show how they could be applied to a tree structure like the file system. I hope it was helpful!

Up to this point in the series all the examples have been very concrete; we have implemented custom folds for each domain we have encountered. Can we be a bit more generic and build some reusable fold implementations?

In the next post we'll look at generic recursive types, and how to work with them.

The source code for this post is available at this gist.

Generic recursive types

This post is the fifth in a series.

In the previous post, we spent some time understanding folds for specific domain types.

In this post, we'll broaden our horizons and look at how to use generic recursive types.

Series contents

Here's the contents of this series:

- **Part 1: Introduction to recursive types and catamorphisms**
  - A simple recursive type
  - Parameterize all the things
  - Introducing catamorphisms
  - Benefits of catamorphisms
  - Rules for creating a catamorphism
- **Part 2: Catamorphism examples**
  - Catamorphism example: File system domain
  - Catamorphism example: Product domain
- **Part 3: Introducing folds**
  - A flaw in our catamorphism implementation
  - Introducing `fold`
  - Problems with fold
  - Using functions as accumulators
  - Introducing `foldback`
  - Rules for creating a fold
- **Part 4: Understanding folds**
  - Iteration vs. recursion
  - Fold example: File system domain
  - Common questions about "fold"
- **Part 5: Generic recursive types**
  - LinkedList: A generic recursive type
  - Making the Gift domain generic
  - Defining a generic Container type
  - A third way to implement the gift domain
  - Abstract or concrete? Comparing the three designs
- **Part 6: Trees in the real world**

- Defining a generic Tree type
- The Tree type in the real world
- Mapping the Tree type
- Example: Creating a directory listing
- Example: A parallel grep
- Example: Storing the file system in a database
- Example: Serializing a Tree to JSON
- Example: Deserializing a Tree from JSON
- Example: Deserializing a Tree from JSON - with error handling

---

LinkedList: A generic recursive type

Here's a question: if you only have algebraic types, and you can only combine them as products (tuples, records) or sums (discriminated unions), then how can you make a list type just by using these operations?

The answer is, of course, recursion!

Let's start with the most basic recursive type: the list.

I'm going to call my version `LinkedList`, but it is basically the same as the `list` type in F#.

So, how do you define a list in a recursive way?

Well, it's either empty, or it consists of an element plus another list. In other words we can define it as a choice type ("discriminated union") like this:

```
type LinkedList<'a> =
    | Empty
    | Cons of head:'a * tail:LinkedList<'a>
```

The `Empty` case represents an empty list. The `Cons` case has a tuple: the head element, and the tail, which is another list.

We can then define a particular `LinkedList` value like this:

```
let linkedList = Cons (1, Cons (2, Cons(3, Empty)))
```

Using the native F# list type, the equivalent definition would be:

```
let linkedList = 1 :: 2 :: 3 :: []
```

which is just `[1; 2; 3]`

`cata` for LinkedList

Following the rules in the first post in this series, we can mechanically create a `cata` function by replacing `Empty` and `Cons` with `fEmpty` and `fCons` :

```
module LinkedList =

    let rec cata fCons fEmpty list :'r=
        let recurse = cata fCons fEmpty
        match list with
        | Empty ->
            fEmpty
        | Cons (element,list) ->
            fCons element (recurse list)
```

Note: We will be putting all the functions associated with `LinkedList<'a>` in a module called `LinkedList`. One nice thing about using generic types is that the type name does not clash with a similar module name!

As always, the signatures of the case handling functions are parallel to the signatures of the type constructors, with `LinkedList` replaced by `'r`.

```
val cata :
    fCons:('a -> 'r -> 'r) ->
    fEmpty:'r ->
    list:LinkedList<'a>
    -> 'r
```

`fold` for LinkedList

We can also create a top-down iterative `fold` function using the rules in the earlier post.

```
module LinkedList =

    let rec cata ...

    let rec foldWithEmpty fCons fEmpty acc list :'r=
        let recurse = foldWithEmpty fCons fEmpty
        match list with
        | Empty ->
            fEmpty acc
        | Cons (element,list) ->
            let newAcc = fCons acc element
            recurse newAcc list
```

This `foldWithEmpty` function is not quite the same as the standard `List.fold` function, because it has an extra function parameter for the empty case (`fEmpty`). However, if we eliminate that parameter and just return the accumulator we get this variant:

```
module LinkedList =

    let rec fold fCons acc list :'r=
        let recurse = fold fCons
        match list with
        | Empty ->
            acc
        | Cons (element,list) ->
            let newAcc = fCons acc element
            recurse newAcc list
```

If we compare the signature with the List.fold documentation we can see that they are equivalent, with `'State` replaced by `'r` and `'T list` replaced by `LinkedList<'a>` :

```
LinkedList.fold : ('r     -> 'a -> 'r     ) -> 'r      -> LinkedList<'a> -> 'r
List.fold       : ('State -> 'T -> 'State) -> 'State -> 'T list         -> 'State
```

Let's test that `fold` works by doing a small sum:

```
let linkedList = Cons (1, Cons (2, Cons(3, Empty)))
linkedList |> LinkedList.fold (+) 0
// Result => 6
```

`foldBack` for LinkedList

Finally we can create a `foldBack` function, using the "function accumulator" approach described in the previous post:

```
module LinkedList =

    let rec cata ...

    let rec fold ...

    let foldBack fCons list acc :'r=
        let fEmpty' generator =
            generator acc
        let fCons' generator element=
            fun innerResult ->
                let newResult = fCons element innerResult
                generator newResult
        let initialGenerator = id
        foldWithEmpty fCons' fEmpty' initialGenerator  list
```

Again, if we compare the signature with the List.foldBack documentation, they are also equivalent, with `'State` replaced by `'r` and `'T list` replaced by `LinkedList<'a>` :

```
LinkedList.foldBack : ('a -> 'r     -> 'r    ) -> LinkedList<'a> -> 'r     -> 'r
List.foldBack       : ('T -> 'State -> 'State) -> 'T list        -> 'State -> 'State
```

Using `foldBack` to convert between list types

In the first post we noted that catamorphisms could be used for converting between types of similar structure.

Let's demonstrate that now by creating some functions that convert from `LinkedList` to the native `list` type and back again.

To convert a `LinkedList` to a native `list` all we need to do is replace `Cons` with `::` and `Empty` with `[]` :

```
module LinkedList =

    let toList linkedList =
        let fCons head tail = head::tail
        let initialState = []
        foldBack fCons linkedList initialState
```

To convert the other way, we need to replace `::` with `Cons` and `[]` with `Empty` :

```
module LinkedList =

    let ofList list =
        let fCons head tail = Cons(head,tail)
        let initialState = Empty
        List.foldBack fCons list initialState
```

Simple! Let's test `toList`:

```
let linkedList = Cons (1, Cons (2, Cons(3, Empty)))
linkedList |> LinkedList.toList
// Result => [1; 2; 3]
```

and `ofList`:

```
let list = [1;2;3]
list |> LinkedList.ofList
// Result => Cons (1,Cons (2,Cons (3,Empty)))
```

Both work as expected.

Using `foldBack` to implement other functions

I said earlier that a catamorphism function (for linear lists, `foldBack`) is the most basic function available for a recursive type, and in fact is the *only* function you need!

Let's see for ourselves by implementing some other common functions using `foldBack`.

Here's `map` defined in terms of `foldBack`:

```
module LinkedList =

    /// map a function "f" over all elements
    let map f list =
        // helper function
        let folder head tail =
            Cons(f head,tail)

        foldBack folder list Empty
```

And here's a test:

```
let linkedList = Cons (1, Cons (2, Cons(3, Empty)))

linkedList |> LinkedList.map (fun i -> i+10)
// Result => Cons (11,Cons (12,Cons (13,Empty)))
```

Here's `filter` defined in terms of `foldBack`:

```
module LinkedList =

    /// return a new list of elements for which "pred" is true
    let filter pred list =
        // helper function
        let folder head tail =
            if pred head then
                Cons(head,tail)
            else
                tail

        foldBack folder list Empty
```

And here's a test:

```
let isOdd n = (n%2=1)
let linkedList = Cons (1, Cons (2, Cons(3, Empty)))

linkedList |> LinkedList.filter isOdd
// Result => Cons (1,Cons (3,Empty))
```

Finally, here's `rev` defined in terms of `fold`:

```
/// reverse the elements of the list
let rev list =
    // helper function
    let folder tail head =
        Cons(head,tail)

    fold folder Empty list
```

And here's a test:

```
let linkedList = Cons (1, Cons (2, Cons(3, Empty)))
linkedList |> LinkedList.rev
// Result => Cons (3,Cons (2,Cons (1,Empty)))
```

So, I hope you're convinced!

Avoiding generator functions

I mentioned earlier that there was an alternative and (sometimes) more efficient way to implement `foldBack` without using generators or continuations.

As we have seen, `foldBack` is reverse iteration, which means that it is the same as `fold` applied to a reversed list!

So we could implement it like this:

```
let foldBack_ViaRev fCons list acc :'r=
    let fCons' acc element =
        // just swap the params!
        fCons element acc
    list
    |> rev
    |> fold fCons' acc
```

It involves making an extra copy of the list, but on the other hand there is no longer a large set of pending continuations. It might be worth comparing the profile of the two versions in your environment if performance is an issue.

Making the Gift domain generic

In the rest of this post, we'll look at the `Gift` type and see if we can make it more generic.

As a reminder, here is the original design:

```
type Gift =
    | Book of Book
    | Chocolate of Chocolate
    | Wrapped of Gift * WrappingPaperStyle
    | Boxed of Gift
    | WithACard of Gift * message:string
```

Three of the cases are recursive and two are non-recursive.

Now, the focus of this particular design was on modelling the domain, which is why there are so many separate cases.

But if we want to focus on *reusability* instead of domain modelling, then we should simplify the design to the essentials, and all these special cases now become a hindrance.

To make this ready for reuse, then, let's collapse all the non-recursive cases into one case, say `GiftContents`, and all the recursive cases into another case, say `GiftDecoration`, like this:

```
// unified data for non-recursive cases
type GiftContents =
    | Book of Book
    | Chocolate of Chocolate

// unified data for recursive cases
type GiftDecoration =
    | Wrapped of WrappingPaperStyle
    | Boxed
    | WithACard of string

type Gift =
    // non-recursive case
    | Contents of GiftContents
    // recursive case
    | Decoration of Gift * GiftDecoration
```

The main `Gift` type has only two cases now: the non-recursive one and the recursive one.

Defining a generic Container type

Now that the type is simplified, we can "genericize" it by allowing *any* kind of contents *and* any kind of decoration.

```
type Container<'ContentData,'DecorationData> =
    | Contents of 'ContentData
    | Decoration of 'DecorationData * Container<'ContentData,'DecorationData>
```

And as before, we can mechanically create a `cata` and `fold` and `foldBack` for it, using the standard process:

```
module Container =

    let rec cata fContents fDecoration (container:Container<'ContentData,'DecorationDa
ta>) :'r =
        let recurse = cata fContents fDecoration
        match container with
        | Contents contentData ->
            fContents contentData
        | Decoration (decorationData,subContainer) ->
            fDecoration decorationData (recurse subContainer)
```

```
    (*
    val cata :
        // function parameters
        fContents:('ContentData -> 'r) ->
        fDecoration:('DecorationData -> 'r -> 'r) ->
        // input
        container:Container<'ContentData,'DecorationData> ->
        // return value
        'r
    *)

    let rec fold fContents fDecoration acc (container:Container<'ContentData,'Decorati
onData>) :'r =
        let recurse = fold fContents fDecoration
        match container with
        | Contents contentData ->
            fContents acc contentData
        | Decoration (decorationData,subContainer) ->
            let newAcc = fDecoration acc decorationData
            recurse newAcc subContainer

    (*
    val fold :
        // function parameters
        fContents:('a -> 'ContentData -> 'r) ->
        fDecoration:('a -> 'DecorationData -> 'a) ->
        // accumulator
        acc:'a ->
        // input
        container:Container<'ContentData,'DecorationData> ->
        // return value
        'r
    *)

    let foldBack fContents fDecoration (container:Container<'ContentData,'DecorationDa
ta>) :'r =
        let fContents' generator contentData =
            generator (fContents contentData)
        let fDecoration' generator decorationData =
            let newGenerator innerValue =
                let newInnerValue = fDecoration decorationData innerValue
                generator newInnerValue
            newGenerator
        fold fContents' fDecoration' id container

    (*
    val foldBack :
        // function parameters
        fContents:('ContentData -> 'r) ->
        fDecoration:('DecorationData -> 'r -> 'r) ->
        // input
        container:Container<'ContentData,'DecorationData> ->
        // return value
```

```
        'r
    *)
```

Converting the gift domain to use the Container type

Let's convert the gift type to this generic Container type:

```
type Gift = Container<GiftContents,GiftDecoration>
```

Now we need some helper methods to construct values while hiding the "real" cases of the generic type:

```
let fromBook book =
    Contents (Book book)

let fromChoc choc =
    Contents (Chocolate choc)

let wrapInPaper paperStyle innerGift =
    let container = Wrapped paperStyle
    Decoration (container, innerGift)

let putInBox innerGift =
    let container = Boxed
    Decoration (container, innerGift)

let withCard message innerGift =
    let container = WithACard message
    Decoration (container, innerGift)
```

Finally we can create some test values:

```
let wolfHall = {title="Wolf Hall"; price=20m}
let yummyChoc = {chocType=SeventyPercent; price=5m}

let birthdayPresent =
    wolfHall
    |> fromBook
    |> wrapInPaper HappyBirthday
    |> withCard "Happy Birthday"

let christmasPresent =
    yummyChoc
    |> fromChoc
    |> putInBox
    |> wrapInPaper HappyHolidays
```

The `totalCost` function using the Container type

The "total cost" function can be written using `fold`, since it doesn't need any inner data.

Unlike the earlier implementations, we only have two function parameters, `fContents` and `fDecoration`, so each of these will need some pattern matching to get at the "real" data.

Here's the code:

```
let totalCost gift =

    let fContents costSoFar contentData =
        match contentData with
        | Book book ->
            costSoFar + book.price
        | Chocolate choc ->
            costSoFar + choc.price

    let fDecoration costSoFar decorationInfo =
        match decorationInfo with
        | Wrapped style ->
            costSoFar + 0.5m
        | Boxed ->
            costSoFar + 1.0m
        | WithACard message ->
            costSoFar + 2.0m

    // initial accumulator
    let initialAcc = 0m

    // call the fold
    Container.fold fContents fDecoration initialAcc gift
```

And the code works as expected:

```
birthdayPresent |> totalCost
// 22.5m

christmasPresent |> totalCost
// 6.5m
```

The `description` function using the Container type

The "description" function needs to be written using `foldBack`, since it *does* need the inner data. As with the code above, we need some pattern matching to get at the "real" data for each case.

```
let description gift =

    let fContents contentData =
        match contentData with
        | Book book ->
            sprintf "'%s'" book.title
        | Chocolate choc ->
            sprintf "%A chocolate" choc.chocType

    let fDecoration decorationInfo innerText =
        match decorationInfo with
        | Wrapped style ->
            sprintf "%s wrapped in %A paper" innerText style
        | Boxed ->
            sprintf "%s in a box" innerText
        | WithACard message ->
            sprintf "%s with a card saying '%s'" innerText message

    // main call
    Container.foldBack fContents fDecoration gift
```

And again the code works as we want:

```
birthdayPresent |> description
// CORRECT "'Wolf Hall' wrapped in HappyBirthday paper with a card saying 'Happy Birth
day'"

christmasPresent |> description
// CORRECT "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"
```

A third way to implement the gift domain

That all looks quite nice, doesn't it?

But I have to confess that I have been holding something back.

None of that code above was strictly necessary, because it turns out that there is yet *another* way to model a `Gift`, without creating any new generic types at all!

The `Gift` type is basically a linear sequence of decorations, with some content as the final step. We can just model this as a pair -- a `Content` and a list of `Decoration`. Or to make it a little friendlier, a record with two fields: one for the content and one for the decorations.

```
type Gift = {contents: GiftContents; decorations: GiftDecoration list}
```

That's it! No other new types needed!

Building values using the record type

As before, let's create some helpers to construct values using this type:

```
let fromBook book =
    { contents = (Book book); decorations = [] }

let fromChoc choc =
    { contents = (Chocolate choc); decorations = [] }

let wrapInPaper paperStyle innerGift =
    let decoration = Wrapped paperStyle
    { innerGift with decorations = decoration::innerGift.decorations }

let putInBox innerGift =
    let decoration = Boxed
    { innerGift with decorations = decoration::innerGift.decorations }

let withCard message innerGift =
    let decoration = WithACard message
    { innerGift with decorations = decoration::innerGift.decorations }
```

With these helper functions, the way the values are constructed is *identical* to the previous version. This is why it is good to hide your raw constructors, folks!

```
let wolfHall = {title="Wolf Hall"; price=20m}
let yummyChoc = {chocType=SeventyPercent; price=5m}

let birthdayPresent =
    wolfHall
    |> fromBook
    |> wrapInPaper HappyBirthday
    |> withCard "Happy Birthday"

let christmasPresent =
    yummyChoc
    |> fromChoc
    |> putInBox
    |> wrapInPaper HappyHolidays
```

The `totalCost` function using the record type

The `totalCost` function is even easier to write now.

```
let totalCost gift =

    let contentCost =
        match gift.contents with
        | Book book ->
            book.price
        | Chocolate choc ->
            choc.price

    let decorationFolder costSoFar decorationInfo =
        match decorationInfo with
        | Wrapped style ->
            costSoFar + 0.5m
        | Boxed ->
            costSoFar + 1.0m
        | WithACard message ->
            costSoFar + 2.0m

    let decorationCost =
        gift.decorations |> List.fold decorationFolder 0m

    // total cost
    contentCost + decorationCost
```

The `description` function using the record type

Similarly, the `description` function is also easy to write.

```
let description gift =

    let contentDescription =
        match gift.contents with
        | Book book ->
            sprintf "'%s'" book.title
        | Chocolate choc ->
            sprintf "%A chocolate" choc.chocType

    let decorationFolder decorationInfo innerText =
        match decorationInfo with
        | Wrapped style ->
            sprintf "%s wrapped in %A paper" innerText style
        | Boxed ->
            sprintf "%s in a box" innerText
        | WithACard message ->
            sprintf "%s with a card saying '%s'" innerText message

    List.foldBack decorationFolder gift.decorations contentDescription
```

Abstract or concrete? Comparing the three designs

If you are confused by this plethora of designs, I don't blame you!

But as it happens, the three different definitions are actually interchangable:

The original version

```
type Gift =
    | Book of Book
    | Chocolate of Chocolate
    | Wrapped of Gift * WrappingPaperStyle
    | Boxed of Gift
    | WithACard of Gift * message:string
```

The generic container version

```
type Container<'ContentData,'DecorationData> =
    | Contents of 'ContentData
    | Decoration of 'DecorationData * Container<'ContentData,'DecorationData>

type GiftContents =
    | Book of Book
    | Chocolate of Chocolate

type GiftDecoration =
    | Wrapped of WrappingPaperStyle
    | Boxed
    | WithACard of string

type Gift = Container<GiftContents,GiftDecoration>
```

The record version

```
type GiftContents =
    | Book of Book
    | Chocolate of Chocolate

type GiftDecoration =
    | Wrapped of WrappingPaperStyle
    | Boxed
    | WithACard of string

type Gift = {contents: GiftContents; decorations: GiftDecoration list}
```

If this is not obvious, it might be helpful to read my post on data type sizes. It explains how two types can be "equivalent", even though they appear to be completely different at first glance.

Picking a design

So which design is best? The answer is, as always, "it depends".

For modelling and documenting a domain, I like the first design with the five explicit cases. Being easy for other people to understand is more important to me than introducing abstraction for the sake of reusability.

If I wanted a reusable model that was applicable in many situations, I'd probably choose the second ("Container") design. It seems to me that this type does represent a commonly encountered situation, where the contents are one kind of thing and the wrappers are another kind of thing. This abstraction is therefore likely to get some use.

The final "pair" model is fine, but by separating the two components, we've over-abstracted the design for this scenario. In other situations, this design might be a great fit (e.g. the decorator pattern), but not here, in my opinion.

There is one further choice which gives you the best of all worlds.

As I noted above, all the designs are logically equivalent, which means there are "lossless" mappings between them. In that case, your "public" design can be the domain-oriented one, like the first one, but behind the scenes you can map it to a more efficient and reusable "private" type.

Even the F# list implementation itself does this. For example, some of the functions in the `List` module, such `foldBack` and `sort`, convert the list into an array, do the operations, and then convert it back to a list again.

Summary

In this post we looked at some ways of modelling the `Gift` as a generic type, and the pros and cons of each approach.

In the next post we'll look at real-world examples of using a generic recursive type.

The source code for this post is available at this gist.

Trees in the real world

This post is the sixth in a series.

In the previous post, we briefly looked at some generic types.

In this post, we'll do some deeper dives into some real-world examples of using trees and folds.

Series contents

Here's the contents of this series:

- **Part 1: Introduction to recursive types and catamorphisms**
  - A simple recursive type
  - Parameterize all the things
  - Introducing catamorphisms
  - Benefits of catamorphisms
  - Rules for creating a catamorphism
- **Part 2: Catamorphism examples**
  - Catamorphism example: File system domain
  - Catamorphism example: Product domain
- **Part 3: Introducing folds**
  - A flaw in our catamorphism implementation
  - Introducing `fold`
  - Problems with fold
  - Using functions as accumulators
  - Introducing `foldback`
  - Rules for creating a fold
- **Part 4: Understanding folds**
  - Iteration vs. recursion
  - Fold example: File system domain
  - Common questions about "fold"
- **Part 5: Generic recursive types**
  - LinkedList: A generic recursive type
  - Making the Gift domain generic
  - Defining a generic Container type
  - A third way to implement the gift domain
  - Abstract or concrete? Comparing the three designs

- **Part 6: Trees in the real world**
  - Defining a generic Tree type
  - The Tree type in the real world
  - Mapping the Tree type
  - Example: Creating a directory listing
  - Example: A parallel grep
  - Example: Storing the file system in a database
  - Example: Serializing a Tree to JSON
  - Example: Deserializing a Tree from JSON
  - Example: Deserializing a Tree from JSON - with error handling

---

Defining a generic Tree type

In this post, we'll be working with a generic `Tree` inspired by the `FileSystem` domain that we explored earlier.

Here was the original design:

```
type FileSystemItem =
    | File of FileInfo
    | Directory of DirectoryInfo
and FileInfo = {name:string; fileSize:int}
and DirectoryInfo = {name:string; dirSize:int; subitems:FileSystemItem list}
```

We can separate out the data from the recursion, and create a generic `Tree` type like this:

```
type Tree<'LeafData,'INodeData> =
    | LeafNode of 'LeafData
    | InternalNode of 'INodeData * Tree<'LeafData,'INodeData> seq
```

Notice that I have used `seq` to represent the subitems rather than `list`. The reason for this will become apparent shortly.

The file system domain can then be modelled using `Tree` by specifying `FileInfo` as data associated with a leaf node and `DirectoryInfo` as data associated with an internal node:

```
type FileInfo = {name:string; fileSize:int}
type DirectoryInfo = {name:string; dirSize:int}

type FileSystemItem = Tree<FileInfo,DirectoryInfo>
```

`cata` and `fold` for Tree

We can define `cata` and `fold` in the usual way:

```
module Tree =

    let rec cata fLeaf fNode (tree:Tree<'LeafData,'INodeData>) :'r =
        let recurse = cata fLeaf fNode
        match tree with
        | LeafNode leafInfo ->
            fLeaf leafInfo
        | InternalNode (nodeInfo,subtrees) ->
            fNode nodeInfo (subtrees |> Seq.map recurse)

    let rec fold fLeaf fNode acc (tree:Tree<'LeafData,'INodeData>) :'r =
        let recurse = fold fLeaf fNode
        match tree with
        | LeafNode leafInfo ->
            fLeaf acc leafInfo
        | InternalNode (nodeInfo,subtrees) ->
            // determine the local accumulator at this level
            let localAccum = fNode acc nodeInfo
            // thread the local accumulator through all the subitems using Seq.fold
            let finalAccum = subtrees |> Seq.fold recurse localAccum
            // ... and return it
            finalAccum
```

Note that I am *not* going to implement `foldBack` for the `Tree` type, because it's unlikely that the tree will get so deep as to cause a stack overflow. Functions that need inner data can use `cata`.

Modelling the File System domain with Tree

Let's test it with the same values that we used before:

```
let fromFile (fileInfo:FileInfo) =
    LeafNode fileInfo

let fromDir (dirInfo:DirectoryInfo) subitems =
    InternalNode (dirInfo,subitems)

let readme = fromFile {name="readme.txt"; fileSize=1}
let config = fromFile {name="config.xml"; fileSize=2}
let build  = fromFile {name="build.bat"; fileSize=3}
let src = fromDir {name="src"; dirSize=10} [readme; config; build]
let bin = fromDir {name="bin"; dirSize=10} []
let root = fromDir {name="root"; dirSize=5} [src; bin]
```

The `totalSize` function is almost identical to the one in the previous post:

```
let totalSize fileSystemItem =
    let fFile acc (file:FileInfo) =
        acc + file.fileSize
    let fDir acc (dir:DirectoryInfo)=
        acc + dir.dirSize
    Tree.fold fFile fDir 0 fileSystemItem

readme |> totalSize  // 1
src |> totalSize     // 16 = 10 + (1 + 2 + 3)
root |> totalSize    // 31 = 5 + 16 + 10
```

And so is the `largestFile` function:

```
let largestFile fileSystemItem =
    let fFile (largestSoFarOpt:FileInfo option) (file:FileInfo) =
        match largestSoFarOpt with
        | None ->
            Some file
        | Some largestSoFar ->
            if largestSoFar.fileSize > file.fileSize then
                Some largestSoFar
            else
                Some file

    let fDir largestSoFarOpt dirInfo =
        largestSoFarOpt

    // call the fold
    Tree.fold fFile fDir None fileSystemItem

readme |> largestFile
// Some {name = "readme.txt"; fileSize = 1}

src |> largestFile
// Some {name = "build.bat"; fileSize = 3}

bin |> largestFile
// None

root |> largestFile
// Some {name = "build.bat"; fileSize = 3}
```

The source code for this section is available at this gist.

The Tree type in the real world

We can use the `Tree` to model the *real* file system too! To do this, just set the leaf node type to `System.IO.FileInfo` and the internal node type to `System.IO.DirectoryInfo`.

```
open System
open System.IO

type FileSystemTree = Tree<IO.FileInfo,IO.DirectoryInfo>
```

And let's create some helper methods to create the various nodes:

```
let fromFile (fileInfo:FileInfo) =
    LeafNode fileInfo

let rec fromDir (dirInfo:DirectoryInfo) =
    let subItems = seq{
        yield! dirInfo.EnumerateFiles() |> Seq.map fromFile
        yield! dirInfo.EnumerateDirectories() |> Seq.map fromDir
        }
    InternalNode (dirInfo,subItems)
```

Now you can see why I used `seq` rather than `list` for the subitems. The `seq` is lazy, which means that we can create nodes without actually hitting the disk.

Here's the `totalSize` function again, this time using the real file information:

```
let totalSize fileSystemItem =
    let fFile acc (file:FileInfo) =
        acc + file.Length
    let fDir acc (dir:DirectoryInfo)=
        acc
    Tree.fold fFile fDir 0L fileSystemItem
```

Let's see what the size of the current directory is:

```
// set the current directory to the current source directory
Directory.SetCurrentDirectory __SOURCE_DIRECTORY__

// get the current directory as a Tree
let currentDir = fromDir (DirectoryInfo("."))

// get the size of the current directory
currentDir  |> totalSize
```

Similarly, we can get the largest file:

```
let largestFile fileSystemItem =
    let fFile (largestSoFarOpt:FileInfo option) (file:FileInfo) =
        match largestSoFarOpt with
        | None ->
            Some file
        | Some largestSoFar ->
            if largestSoFar.Length > file.Length then
                Some largestSoFar
            else
                Some file

    let fDir largestSoFarOpt dirInfo =
        largestSoFarOpt

    // call the fold
    Tree.fold fFile fDir None fileSystemItem

currentDir |> largestFile
```

So that's one big benefit of using generic recursive types. If we can turn a real-world hierarchy into our tree structure, we can get all the benefits of fold "for free".

Mapping with generic types

One other advantage of using generic types is that you can do things like `map` -- converting every element to a new type without changing the structure.

We can see this in action with the real world file system. But first we need to define `map` for the `Tree` type!

The implementation of `map` can also be done mechanically, using the following rules:

- Create a function parameter to handle each case in the structure.
- For non-recursive cases
    - First, use the function parameter to transform the non-recursive data associated with that case
    - Then wrap the result in the same case constructor
- For recursive cases, perform two steps:
    - First, use the function parameter to transform the non-recursive data associated with that case
    - Next, recursively `map` the nested values.
    - Finally, wrap the results in the same case constructor

Here's the implementation of `map` for `Tree`, created by following those rules:

```
module Tree =

    let rec cata ...

    let rec fold ...

    let rec map fLeaf fNode (tree:Tree<'LeafData,'INodeData>) =
        let recurse = map fLeaf fNode
        match tree with
        | LeafNode leafInfo ->
            let newLeafInfo = fLeaf leafInfo
            LeafNode newLeafInfo
        | InternalNode (nodeInfo,subtrees) ->
            let newNodeInfo = fNode nodeInfo
            let newSubtrees = subtrees |> Seq.map recurse
            InternalNode (newNodeInfo, newSubtrees)
```

If we look at the signature of `Tree.map`, we can see that all the leaf data is transformed to type `'a`, all the node data is transformed to type `'b`, and the final result is a `Tree<'a,'b>`.

```
val map :
  fLeaf:('LeafData -> 'a) ->
  fNode:('INodeData -> 'b) ->
  tree:Tree<'LeafData,'INodeData> ->
  Tree<'a,'b>
```

We can define `Tree.iter` in a similar way:

```
module Tree =

    let rec map ...

    let rec iter fLeaf fNode (tree:Tree<'LeafData,'INodeData>) =
        let recurse = iter fLeaf fNode
        match tree with
        | LeafNode leafInfo ->
            fLeaf leafInfo
        | InternalNode (nodeInfo,subtrees) ->
            subtrees |> Seq.iter recurse
            fNode nodeInfo
```

Example: Creating a directory listing

Let's say we want to use `map` to transform the file system into a directory listing - a tree of strings where each string has information about the corresponding file or directory. Here's how we could do it:

```
let dirListing fileSystemItem =
    let printDate (d:DateTime) = d.ToString()
    let mapFile (fi:FileInfo) =
        sprintf "%10i  %s  %-s"  fi.Length (printDate fi.LastWriteTime) fi.Name
    let mapDir (di:DirectoryInfo) =
        di.FullName
    Tree.map mapFile mapDir fileSystemItem
```

And then we can print the strings out like this:

```
currentDir
|> dirListing
|> Tree.iter (printfn "%s") (printfn "\n%s")
```

The results will look something like this:

```
  8315  10/08/2015 23:37:41  Fold.fsx
  3680  11/08/2015 23:59:01  FoldAndRecursiveTypes.fsproj
  1010  11/08/2015 01:19:07  FoldAndRecursiveTypes.sln
  1107  11/08/2015 23:59:01  HtmlDom.fsx
    79  11/08/2015 01:21:54  LinkedList.fsx
```

The source code for this example is available at this gist.

Example: Creating a parallel grep

Let's look at a more complex example. I'll demonstrate how to create a parallel "grep" style search using `fold`.

The logic will be like this:

- Use `fold` to iterate through the files.
- For each file, if its name doesn't match the desired file pattern, return `None`.
- If the file is to be processed, then return an async that returns all the line matches in the file.
- Next, all these asyncs -- the output of the fold -- are aggregated into a sequence.
- The sequence of asyncs is transformed into a single one using `Async.Parallel` which returns a list of results.

Before we start writing the main code, we'll need some helper functions.

First, a generic function that folds over the lines in a file asynchronously. This will be the basis of the pattern matching.

```
/// Fold over the lines in a file asynchronously
/// passing in the current line and line number tothe folder function.
///
/// Signature:
///   folder:('a -> int -> string -> 'a) ->
///   acc:'a ->
///   fi:FileInfo ->
///   Async<'a>
let foldLinesAsync folder acc (fi:FileInfo) =
    async {
        let mutable acc = acc
        let mutable lineNo = 1
        use sr = new StreamReader(path=fi.FullName)
        while not sr.EndOfStream do
            let! lineText = sr.ReadLineAsync() |> Async.AwaitTask
            acc <- folder acc lineNo lineText
            lineNo <- lineNo + 1
        return acc
    }
```

Next, a little helper that allows us to `map` over `Async` values:

```
let asyncMap f asyncX = async {
    let! x = asyncX
    return (f x)  }
```

Now for the central logic. We will create a function that, given a `textPattern` and a `FileInfo`, will return a list of lines that match the textPattern, but asynchronously:

```
/// return the matching lines in a file, as an async<string list>
let matchPattern textPattern (fi:FileInfo) =
    // set up the regex
    let regex = Text.RegularExpressions.Regex(pattern=textPattern)

    // set up the function to use with "fold"
    let folder results lineNo lineText =
        if regex.IsMatch lineText then
            let result = sprintf "%40s:%-5i   %s" fi.Name lineNo lineText
            result :: results
        else
            // pass through
            results

    // main flow
    fi
    |> foldLinesAsync folder []
    // the fold output is in reverse order, so reverse it
    |> asyncMap List.rev
```

And now for the `grep` function itself:

```
let grep filePattern textPattern fileSystemItem =
    let regex = Text.RegularExpressions.Regex(pattern=filePattern)

    /// if the file matches the pattern
    /// do the matching and return Some async, else None
    let matchFile (fi:FileInfo) =
        if regex.IsMatch fi.Name then
            Some (matchPattern textPattern fi)
        else
            None

    /// process a file by adding its async to the list
    let fFile asyncs (fi:FileInfo) =
        // add to the list of asyncs
        (matchFile fi) :: asyncs

    // for directories, just pass through the list of asyncs
    let fDir asyncs (di:DirectoryInfo)  =
        asyncs

    fileSystemItem
    |> Tree.fold fFile fDir []    // get the list of asyncs
    |> Seq.choose id              // choose the Somes (where a file was processed)
    |> Async.Parallel             // merge all asyncs into a single async
    |> asyncMap (Array.toList >> List.collect id)  // flatten array of lists into a si
ngle list
```

Let's test it!

```
currentDir
|> grep "fsx" "LinkedList"
|> Async.RunSynchronously
```

The result will look something like this:

```
"                   SizeOfTypes.fsx:120     type LinkedList<'a> = ";
"                   SizeOfTypes.fsx:122         | Cell of head:'a * tail:LinkedList<'a>
";
"                   SizeOfTypes.fsx:125     let S = size(LinkedList<'a>)";
"       RecursiveTypesAndFold-3.fsx:15      // LinkedList";
"       RecursiveTypesAndFold-3.fsx:18      type LinkedList<'a> = ";
"       RecursiveTypesAndFold-3.fsx:20          | Cons of head:'a * tail:LinkedList<'a>
";
"       RecursiveTypesAndFold-3.fsx:26      module LinkedList = ";
"       RecursiveTypesAndFold-3.fsx:39              list:LinkedList<'a> ";
"       RecursiveTypesAndFold-3.fsx:64              list:LinkedList<'a> -> ";
```

That's not bad for about 40 lines of code. This conciseness is because we are using various kinds of `fold` and `map` which hide the recursion, allowing us to focus on the pattern matching logic itself.

Of course, this is not at all efficient or optimized (an async for every line!), and so I wouldn't use it as a real implementation, but it does give you an idea of the power of fold.

The source code for this example is available at this gist.

Example: Storing the file system in a database

For the next example, let's look at how to store a file system tree in a database. I don't really know why you would want to do that, but the principles would work equally well for storing any hierarchical structure, so I will demonstrate it anyway!

To model the file system hierarchy in the database, say that we have four tables:

- `DbDir` stores information about each directory.
- `DbFile` stores information about each file.
- `DbDir_File` stores the relationship between a directory and a file.
- `DbDir_Dir` stores the relationship between a parent directory and a child directory.

Here are the database table definitions:

```
CREATE TABLE DbDir (
    DirId int IDENTITY NOT NULL,
    Name nvarchar(50) NOT NULL
)

CREATE TABLE DbFile (
    FileId int IDENTITY NOT NULL,
    Name nvarchar(50) NOT NULL,
    FileSize int NOT NULL
)

CREATE TABLE DbDir_File (
    DirId int NOT NULL,
    FileId int NOT NULL
)

CREATE TABLE DbDir_Dir (
    ParentDirId int NOT NULL,
    ChildDirId int NOT NULL
)
```

That's simple enough. But note that in order to save a directory completely along with its relationships to its child items, we first need the ids of all its children, and each child directory needs the ids of its children, and so on.

This implies that we should use `cata` instead of `fold`, so that we have access to the data from the lower levels of the hierarchy.

Implementing the database functions

We're not wise enough to be using the SQL Provider and so we have written our own table insertion functions, like this dummy one:

```
/// Insert a DbFile record
let insertDbFile name (fileSize:int64) =
    let id = nextIdentity()
    printfn "%10s: inserting id:%i name:%s size:%i" "DbFile" id name fileSize
```

In a real database, the identity column would be automatically generated for you, but for this example, I'll use a little helper function `nextIdentity`:

```
let nextIdentity =
    let id = ref 0
    fun () ->
        id := !id + 1
        !id

// test
nextIdentity() // 1
nextIdentity() // 2
nextIdentity() // 3
```

Now in order to insert a directory, we need to first know all the ids of the files in the directory. This implies that the `insertDbFile` function should return the id that was generated.

```
/// Insert a DbFile record and return the new file id
let insertDbFile name (fileSize:int64) =
    let id = nextIdentity()
    printfn "%10s: inserting id:%i name:%s size:%i" "DbFile" id name fileSize
    id
```

But that logic applies to the directories too:

```
/// Insert a DbDir record and return the new directory id
let insertDbDir name =
    let id = nextIdentity()
    printfn "%10s: inserting id:%i name:%s" "DbDir" id name
    id
```

But that's still not good enough. When the child ids are passed to the parent directory, it needs to distinguish between files and directories, because the relations are stored in different tables.

No problem -- we'll just use a choice type to distinguish between them!

```
type PrimaryKey =
    | FileId of int
    | DirId of int
```

With this in place, we can complete the implementation of the database functions:

```
/// Insert a DbFile record and return the new PrimaryKey
let insertDbFile name (fileSize:int64) =
    let id = nextIdentity()
    printfn "%10s: inserting id:%i name:%s size:%i" "DbFile" id name fileSize
    FileId id

/// Insert a DbDir record and return the new PrimaryKey
let insertDbDir name =
    let id = nextIdentity()
    printfn "%10s: inserting id:%i name:%s" "DbDir" id name
    DirId id

/// Insert a DbDir_File record
let insertDbDir_File dirId fileId =
    printfn "%10s: inserting parentDir:%i childFile:%i" "DbDir_File" dirId fileId

/// Insert a DbDir_Dir record
let insertDbDir_Dir parentDirId childDirId =
    printfn "%10s: inserting parentDir:%i childDir:%i" "DbDir_Dir" parentDirId childDi
rId
```

Working with the catamorphism

As noted above, we need to use `cata` instead of `fold`, because we need the inner ids at each step.

The function to handle the `File` case is easy -- just insert it and return the `PrimaryKey`.

```
let fFile (fi:FileInfo) =
    insertDbFile fi.Name fi.Length
```

The function to handle the `Directory` case will be passed the `DirectoryInfo` and a sequence of `PrimaryKey`s from the children that have already been inserted.

It should insert the main directory record, then insert the children, and then return the `PrimaryKey` for the next higher level:

```
let fDir (di:DirectoryInfo) childIds  =
    let dirId = insertDbDir di.Name
    // insert the children
    // return the id to the parent
    dirId
```

After inserting the directory record and getting its id, for each child id, we insert either into the `DbDir_File` table or the `DbDir_Dir`, depending on the type of the `childId`.

```
let fDir (di:DirectoryInfo) childIds  =
    let dirId = insertDbDir di.Name
    let parentPK = pkToInt dirId
    childIds |> Seq.iter (fun childId ->
        match childId with
        | FileId fileId -> insertDbDir_File parentPK fileId
        | DirId childDirId -> insertDbDir_Dir parentPK childDirId
    )
    // return the id to the parent
    dirId
```

Note that I've also created a little helper function `pkToInt` that extracts the integer id from the `PrimaryKey` type.

Here is all the code in one chunk:

```
open System
open System.IO

let nextIdentity =
    let id = ref 0
    fun () ->
        id := !id + 1
        !id

type PrimaryKey =
    | FileId of int
    | DirId of int

/// Insert a DbFile record and return the new PrimaryKey
let insertDbFile name (fileSize:int64) =
    let id = nextIdentity()
    printfn "%10s: inserting id:%i name:%s size:%i" "DbFile" id name fileSize
    FileId id

/// Insert a DbDir record and return the new PrimaryKey
let insertDbDir name =
    let id = nextIdentity()
    printfn "%10s: inserting id:%i name:%s" "DbDir" id name
    DirId id

/// Insert a DbDir_File record
let insertDbDir_File dirId fileId =
    printfn "%10s: inserting parentDir:%i childFile:%i" "DbDir_File" dirId fileId

/// Insert a DbDir_Dir record
let insertDbDir_Dir parentDirId childDirId =
    printfn "%10s: inserting parentDir:%i childDir:%i" "DbDir_Dir" parentDirId childDi
rId
```

```
let pkToInt primaryKey =
    match primaryKey with
    | FileId fileId -> fileId
    | DirId dirId -> dirId

let insertFileSystemTree fileSystemItem =

    let fFile (fi:FileInfo) =
        insertDbFile fi.Name fi.Length

    let fDir (di:DirectoryInfo) childIds  =
        let dirId = insertDbDir di.Name
        let parentPK = pkToInt dirId
        childIds |> Seq.iter (fun childId ->
            match childId with
            | FileId fileId -> insertDbDir_File parentPK fileId
            | DirId childDirId -> insertDbDir_Dir parentPK childDirId
        )
        // return the id to the parent
        dirId

    fileSystemItem
    |> Tree.cata fFile fDir
```

Now let's test it:

```
// get the current directory as a Tree
let currentDir = fromDir (DirectoryInfo("."))

// insert into the database
currentDir
|> insertFileSystemTree
```

The output should look something like this:

```
     DbDir: inserting id:41 name:FoldAndRecursiveTypes
    DbFile: inserting id:42 name:Fold.fsx size:8315
DbDir_File: inserting parentDir:41 childFile:42
    DbFile: inserting id:43 name:FoldAndRecursiveTypes.fsproj size:3680
DbDir_File: inserting parentDir:41 childFile:43
    DbFile: inserting id:44 name:FoldAndRecursiveTypes.sln size:1010
DbDir_File: inserting parentDir:41 childFile:44
...
     DbDir: inserting id:57 name:bin
     DbDir: inserting id:58 name:Debug
 DbDir_Dir: inserting parentDir:57 childDir:58
 DbDir_Dir: inserting parentDir:41 childDir:57
```

You can see that the ids are being generated as the files are iterated over, and that each `DbFile` insert is followed by a `DbDir_File` insert.

The source code for this example is available at this gist.

Example: Serializing a Tree to JSON

Let's look at another common challenge: serializing and deserializing a tree to JSON, XML, or some other format.

Let's use the Gift domain again, but this time, we'll model the `Gift` type as a tree. That means we get to put more than one thing in a box!

Modelling the Gift domain as a tree

Here are the main types again, but notice that the final `Gift` type is defined as a tree:

```
type Book = {title: string; price: decimal}
type ChocolateType = Dark | Milk | SeventyPercent
type Chocolate = {chocType: ChocolateType ; price: decimal}

type WrappingPaperStyle =
    | HappyBirthday
    | HappyHolidays
    | SolidColor

// unified data for non-recursive cases
type GiftContents =
    | Book of Book
    | Chocolate of Chocolate

// unified data for recursive cases
type GiftDecoration =
    | Wrapped of WrappingPaperStyle
    | Boxed
    | WithACard of string

type Gift = Tree<GiftContents,GiftDecoration>
```

As usual, we can create some helper functions to assist with constructing a `Gift`:

```
let fromBook book =
    LeafNode (Book book)

let fromChoc choc =
    LeafNode (Chocolate choc)

let wrapInPaper paperStyle innerGift =
    let container = Wrapped paperStyle
    InternalNode (container, [innerGift])

let putInBox innerGift =
    let container = Boxed
    InternalNode (container, [innerGift])

let withCard message innerGift =
    let container = WithACard message
    InternalNode (container, [innerGift])

let putTwoThingsInBox innerGift innerGift2 =
    let container = Boxed
    InternalNode (container, [innerGift;innerGift2])
```

And we can create some sample data:

```
let wolfHall = {title="Wolf Hall"; price=20m}
let yummyChoc = {chocType=SeventyPercent; price=5m}

let birthdayPresent =
    wolfHall
    |> fromBook
    |> wrapInPaper HappyBirthday
    |> withCard "Happy Birthday"

let christmasPresent =
    yummyChoc
    |> fromChoc
    |> putInBox
    |> wrapInPaper HappyHolidays

let twoBirthdayPresents =
    let thing1 = wolfHall |> fromBook
    let thing2 = yummyChoc |> fromChoc
    putTwoThingsInBox thing1 thing2
    |> wrapInPaper HappyBirthday

let twoWrappedPresentsInBox =
    let thing1 = wolfHall |> fromBook |> wrapInPaper HappyHolidays
    let thing2 = yummyChoc |> fromChoc  |> wrapInPaper HappyBirthday
    putTwoThingsInBox thing1 thing2
```

Functions like `description` now need to handle a *list* of inner texts, rather than one. We'll just concat the strings together with an `&` separator:

```
let description gift =

    let fLeaf leafData =
        match leafData with
        | Book book ->
            sprintf "'%s'" book.title
        | Chocolate choc ->
            sprintf "%A chocolate" choc.chocType

    let fNode nodeData innerTexts =
        let innerText = String.concat " & " innerTexts
        match nodeData with
        | Wrapped style ->
            sprintf "%s wrapped in %A paper" innerText style
        | Boxed ->
            sprintf "%s in a box" innerText
        | WithACard message ->
            sprintf "%s with a card saying '%s'" innerText message

    // main call
    Tree.cata fLeaf fNode gift
```

Finally, we can check that the function still works as before, and that multiple items are handled correctly:

```
birthdayPresent |> description
// "'Wolf Hall' wrapped in HappyBirthday paper with a card saying 'Happy Birthday'"

christmasPresent |> description
// "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"

twoBirthdayPresents |> description
// "'Wolf Hall' & SeventyPercent chocolate in a box
//   wrapped in HappyBirthday paper"

twoWrappedPresentsInBox |> description
// "'Wolf Hall' wrapped in HappyHolidays paper
//   & SeventyPercent chocolate wrapped in HappyBirthday paper
//   in a box"
```

Step 1: Defining `GiftDto`

Our `Gift` type consists of many discriminated unions. In my experience, these do not serialize well. In fact, most complex types do not serialize well!

So what I like to do is define DTO types that are explicitly designed to be serialized well. In practice this means that the DTO types are constrained as follows:

- Only record types should be used.
- The record fields should consist only primitive values such as `int`, `string` and `bool`.

By doing this, we also get some other advantages:

We gain control of the serialization output. These kinds of data types are handled the same by most serializers, while "strange" things such as unions can be interpreted differently by different libraries.

We have better control of error handling. My number one rule when dealing with serialized data is "trust no one". It's very common that the data is structured correctly but is invalid for the domain: supposedly non-null strings are null, strings are too long, integers are outside the correct bounds, and so on.

By using DTOs, we can be sure that the deserialization step itself will work. Then, when we convert the DTO to a domain type, we can do proper validation.

So, let's define some DTO types for out domain. Each DTO type will correspond to a domain type, so let's start with `GiftContents`. We'll define a corresponding DTO type called `GiftContentsDto` as follows:

```
[<CLIMutableAttribute>]
type GiftContentsDto = {
    discriminator : string // "Book" or "Chocolate"
    // for "Book" case only
    bookTitle: string
    // for "Chocolate" case only
    chocolateType : string // one of "Dark" "Milk" "SeventyPercent"
    // for all cases
    price: decimal
    }
```

Obviously, this quite different from the original `GiftContents`, so let's look at the differences:

- First, it has the `CLIMutableAttribute`, which allows deserializers to construct them using reflection.
- Second, it has a `discriminator` which indicates which case of the original union type is being used. Obviously, this string could be set to anything, so when converting from the DTO back to the domain type, we'll have to check that carefully!
- Next is a series of fields, one for every possible item of data that needs to be stored. For example, in the `Book` case, we need a `bookTitle`, while in the `Chocolate` case, we need the chocolate type. And finally the `price` field which is in both types. Note that the chocolate type is stored as a string as well, and so will also need special treatment

when we convert from DTO to domain.

The `GiftDecorationDto` type is created in the same way, with a discriminator and strings rather than unions.

```
[<CLIMutableAttribute>]
type GiftDecorationDto = {
    discriminator: string // "Wrapped" or "Boxed" or "WithACard"
    // for "Wrapped" case only
    wrappingPaperStyle: string  // "HappyBirthday" or "HappyHolidays" or "SolidColor"

    // for "WithACard" case only
    message: string
    }
```

Finally, we can define a `GiftDto` type as being a tree that is composed of the two DTO types:

```
type GiftDto = Tree<GiftContentsDto,GiftDecorationDto>
```

Step 2: Transforming a `Gift` to a `GiftDto`

Now that we have this DTO type, all we need to do is use `Tree.map` to convert from a `Gift` to a `GiftDto`. And in order to do that, we need to create two functions: one that converts from `GiftContents` to `GiftContentsDto` and one that converts from `GiftDecoration` to `GiftDecorationDto`.

Here's the complete code for `giftToDto`, which should be self-explanatory:

```
let giftToDto (gift:Gift) :GiftDto =

    let fLeaf leafData :GiftContentsDto =
        match leafData with
        | Book book ->
            {discriminator= "Book"; bookTitle=book.title; chocolateType=null; price=bo
ok.price}
        | Chocolate choc ->
            let chocolateType = sprintf "%A" choc.chocType
            {discriminator= "Chocolate"; bookTitle=null; chocolateType=chocolateType;
price=choc.price}

    let fNode nodeData :GiftDecorationDto =
        match nodeData with
        | Wrapped style ->
            let wrappingPaperStyle = sprintf "%A" style
            {discriminator= "Wrapped"; wrappingPaperStyle=wrappingPaperStyle; message=
null}
        | Boxed ->
            {discriminator= "Boxed"; wrappingPaperStyle=null; message=null}
        | WithACard message ->
            {discriminator= "WithACard"; wrappingPaperStyle=null; message=message}

    // main call
    Tree.map fLeaf fNode gift
```

You can see that the case (`Book` , `Chocolate` , etc.) is turned into a `discriminator` string and the `chocolateType` is also turned into a string, just as explained above.

Step 3: Defining a `TreeDto`

I said above that a good DTO should be a record type. Well we have converted the nodes of the tree, but the tree *itself* is a union type! We need to transform the `Tree` type as well, into say a `TreeDto` type.

How can we do this? Just as for the gift DTO types, we will create a record type which contains all the data for both cases. We could use a discriminator field as we did before, but this time, since there are only two choices, leaf and internal node, I'll just check whether the values are null or not when deserializing. If the leaf value is not null, then the record must represent the `LeafNode` case, otherwise the record must represent the `InternalNode` case.

Here's the definition of the data type:

```
/// A DTO that represents a Tree
/// The Leaf/Node choice is turned into a record
[<CLIMutableAttribute>]
type TreeDto<'LeafData,'NodeData> = {
    leafData : 'LeafData
    nodeData : 'NodeData
    subtrees : TreeDto<'LeafData,'NodeData>[] }
```

As before, the type has the `CLIMutableAttribute`. And as before, the type has fields to store the data from all possible choices. The `subtrees` are stored as an array rather than a seq -- this makes the serializer happy!

To create a `TreeDto`, we use our old friend `cata` to assemble the record from a regular `Tree`.

```
/// Transform a Tree into a TreeDto
let treeToDto tree : TreeDto<'LeafData,'NodeData> =

    let fLeaf leafData  =
        let nodeData = Unchecked.defaultof<'NodeData>
        let subtrees = [||]
        {leafData=leafData; nodeData=nodeData; subtrees=subtrees}

    let fNode nodeData subtrees =
        let leafData = Unchecked.defaultof<'NodeData>
        let subtrees = subtrees |> Seq.toArray
        {leafData=leafData; nodeData=nodeData; subtrees=subtrees}

    // recurse to build up the TreeDto
    Tree.cata fLeaf fNode tree
```

Note that in F#, records are not nullable, so I am using `Unchecked.defaultof<'NodeData>` rather than `null` to indicate missing data.

Note also that I am assuming that `LeafData` or `NodeData` are reference types. If `LeafData` or `NodeData` are ever value types like `int` or `bool`, then this approach will break down, because you won't be able to tell the difference between a default value and a missing value. In which case, I'd switch to a discriminator field as before.

Alternatively, I could have used an `IDictionary`. That would be less convenient to deserialize, but would avoid the need for null-checking.

Step 4: Serializing a `TreeDto`

Finally we can serialize the `TreeDto` using a JSON serializer.

For this example, I am using the built-in `DataContractJsonSerializer` so that I don't need to take a dependency on a NuGet package. There are other JSON serializers that might be better for a serious project.

```
#r "System.Runtime.Serialization.dll"

open System.Runtime.Serialization
open System.Runtime.Serialization.Json

let toJson (o:'a) =
    let serializer = new DataContractJsonSerializer(typeof<'a>)
    let encoding = System.Text.UTF8Encoding()
    use stream = new System.IO.MemoryStream()
    serializer.WriteObject(stream,o)
    stream.Close()
    encoding.GetString(stream.ToArray())
```

Step 5: Assembling the pipeline

So, putting it all together, we have the following pipeline:

- Transform `Gift` to `GiftDto` using `giftToDto`,
  that is, use `Tree.map` to go from `Tree<GiftContents,GiftDecoration>` to `Tree<GiftContentsDto,GiftDecorationDto>`
- Transform `Tree` to `TreeDto` using `treeToDto`,
  that is, use `Tree.cata` to go from `Tree<GiftContentsDto,GiftDecorationDto>` to `TreeDto<GiftContentsDto,GiftDecorationDto>`
- Serialize `TreeDto` to a JSON string

Here's some example code:

```
let goodJson = christmasPresent |> giftToDto |> treeToDto |> toJson
```

And here is what the JSON output looks like:

```
{
  "leafData@": null,
  "nodeData@": {
    "discriminator@": "Wrapped",
    "message@": null,
    "wrappingPaperStyle@": "HappyHolidays"
  },
  "subtrees@": [
    {
      "leafData@": null,
      "nodeData@": {
        "discriminator@": "Boxed",
        "message@": null,
        "wrappingPaperStyle@": null
      },
      "subtrees@": [
        {
          "leafData@": {
            "bookTitle@": null,
            "chocolateType@": "SeventyPercent",
            "discriminator@": "Chocolate",
            "price@": 5
          },
          "nodeData@": null,
          "subtrees@": []
        }
      ]
    }
  ]
}
```


The ugly `@` signs on the field names are an artifact of serializing the F# record type. This can be corrected with a bit of effort, but I'm not going to bother right now!

The source code for this example is available at this gist

Example: Deserializing a Tree from JSON

Now that we have created the JSON, what about going the other way and loading it into a `Gift` ?

Simple! We just need to reverse the pipeline:

- Deserialize a JSON string into a `TreeDto` .
- Transform a `TreeDto` into a `Tree` to using `dtoToTree` , that is, go from `TreeDto<GiftContentsDto,GiftDecorationDto>` to

`Tree<GiftContentsDto,GiftDecorationDto>`. We can't use `cata` for this -- we'll have to create a little recursive loop.
- Transform `GiftDto` to `Gift` using `dtoToGift`, that is, use `Tree.map` to go from `Tree<GiftContentsDto,GiftDecorationDto>` to `Tree<GiftContents,GiftDecoration>`.

Step 1: Deserializing a `TreeDto`

We can deserialize the `TreeDto` using a JSON serializer.

```
let fromJson<'a> str =
    let serializer = new DataContractJsonSerializer(typeof<'a>)
    let encoding = System.Text.UTF8Encoding()
    use stream = new System.IO.MemoryStream(encoding.GetBytes(s=str))
    let obj = serializer.ReadObject(stream)
    obj :?> 'a
```

What if the deserialization fails? For now, we will ignore any error handling and let the exception propagate.

Step 2: Transforming a `TreeDto` into a `Tree`

To transform a `TreeDto` into a `Tree` we recursively loop through the record and its subtrees, turning each one into a `InternalNode` or a `LeafNode`, based on whether the appropriate field is null or not.

```
let rec dtoToTree (treeDto:TreeDto<'Leaf,'Node>) :Tree<'Leaf,'Node> =
    let nullLeaf = Unchecked.defaultof<'Leaf>
    let nullNode = Unchecked.defaultof<'Node>

    // check if there is nodeData present
    if treeDto.nodeData <> nullNode then
        if treeDto.subtrees = null then
            failwith "subtrees must not be null if node data present"
        else
            let subtrees = treeDto.subtrees |> Array.map dtoToTree
            InternalNode (treeDto.nodeData,subtrees)
    // check if there is leafData present
    elif treeDto.leafData <> nullLeaf then
        LeafNode (treeDto.leafData)
    // if both missing then fail
    else
        failwith "expecting leaf or node data"
```

As you can see, a number of things could go wrong:

- What if the `leafData` and `nodeData` fields are both null?
- What if the `nodeData` field is not null but the `subtrees` field *is* null?

Again, we will ignore any error handling and just throw exceptions (for now).

Question: Could we create a `cata` for `TreeDto` that would make this code simpler? Would it be worth it?

Step 3: Transforming a `GiftDto` into `Gift`

Now we have a proper tree, we can use `Tree.map` again to convert each leaf and internal node from a DTO to the proper domain type.

That means we need functions that map a `GiftContentsDto` into a `GiftContents` and a `GiftDecorationDto` into a `GiftDecoration`.

Here's the complete code -- it's a lot more complicated than going in the other direction!

The code can be grouped as follows:

- Helper methods (such as `strToChocolateType`) that convert a string into a proper domain type and throw an exception if the input is invalid.
- Case converter methods (such as `bookFromDto`) that convert an entire DTO into a case.
- And finally, the `dtoToGift` function itself. It looks at the `discriminator` field to see which case converter to call, and throws an exception if the discriminator value is not recognized.

```
let strToBookTitle str =
    match str with
    | null -> failwith "BookTitle must not be null"
    | _ -> str

let strToChocolateType str =
    match str with
    | "Dark" -> Dark
    | "Milk" -> Milk
    | "SeventyPercent" -> SeventyPercent
    | _ -> failwithf "ChocolateType %s not recognized" str

let strToWrappingPaperStyle str =
    match str with
    | "HappyBirthday" -> HappyBirthday
    | "HappyHolidays" -> HappyHolidays
    | "SolidColor" -> SolidColor
    | _ -> failwithf "WrappingPaperStyle %s not recognized" str

let strToCardMessage str =
    match str with
```

```
    | null -> failwith "CardMessage must not be null"
    | _ -> str

let bookFromDto (dto:GiftContentsDto) =
    let bookTitle = strToBookTitle dto.bookTitle
    Book {title=bookTitle; price=dto.price}

let chocolateFromDto (dto:GiftContentsDto) =
    let chocType = strToChocolateType dto.chocolateType
    Chocolate {chocType=chocType; price=dto.price}

let wrappedFromDto (dto:GiftDecorationDto) =
    let wrappingPaperStyle = strToWrappingPaperStyle dto.wrappingPaperStyle
    Wrapped wrappingPaperStyle

let boxedFromDto (dto:GiftDecorationDto) =
    Boxed

let withACardFromDto (dto:GiftDecorationDto) =
    let message = strToCardMessage dto.message
    WithACard message

/// Transform a GiftDto to a Gift
let dtoToGift (giftDto:GiftDto) :Gift=

    let fLeaf (leafDto:GiftContentsDto) =
        match leafDto.discriminator with
        | "Book" -> bookFromDto leafDto
        | "Chocolate" -> chocolateFromDto leafDto
        | _ -> failwithf "Unknown leaf discriminator '%s'" leafDto.discriminator

    let fNode (nodeDto:GiftDecorationDto)  =
        match nodeDto.discriminator with
        | "Wrapped" -> wrappedFromDto nodeDto
        | "Boxed" -> boxedFromDto nodeDto
        | "WithACard" -> withACardFromDto nodeDto
        | _ -> failwithf "Unknown node discriminator '%s'" nodeDto.discriminator

    // map the tree
    Tree.map fLeaf fNode giftDto
```

Step 4: Assembling the pipeline

We can now assemble the pipeline that takes a JSON string and creates a `Gift`.

```
let goodGift = goodJson |> fromJson |> dtoToTree |> dtoToGift

// check that the description is unchanged
goodGift |> description
// "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"
```

This works fine, but the error handling is terrible!

Look what happens if we corrupt the JSON a little:

```
let badJson1 = goodJson.Replace("leafData","leafDataXX")

let badJson1_result = badJson1 |> fromJson |> dtoToTree |> dtoToGift
// Exception "The data contract type 'TreeDto' cannot be deserialized because the requ
ired data member 'leafData@' was not found."
```

We get an ugly exception.

Or what if a discriminator is wrong?

```
let badJson2 = goodJson.Replace("Wrapped","Wrapped2")

let badJson2_result = badJson2 |> fromJson |> dtoToTree |> dtoToGift
// Exception "Unknown node discriminator 'Wrapped2'"
```

or one of the values for the WrappingPaperStyle DU?

```
let badJson3 = goodJson.Replace("HappyHolidays","HappyHolidays2")
let badJson3_result = badJson3 |> fromJson |> dtoToTree |> dtoToGift
// Exception "WrappingPaperStyle HappyHolidays2 not recognized"
```

We get lots of exceptions, and as as functional programmers, we should try to remove them whenever we can.

How we can do that will be discussed in the next section.

The source code for this example is available at this gist.

Example: Deserializing a Tree from JSON - with error handling

To address the error handling issue, we're going use the `Result` type shown below:

```
type Result<'a> =
    | Success of 'a
    | Failure of string list
```

I'm not going to explain how it works here. If you are not familar with this approach, please read my post or watch my talk on the topic of functional error handling.

Let's revisit all the steps from the previous section, and use `Result` rather than throwing exceptions.

Step 1: Deserializing a `TreeDto`

When we deserialize the `TreeDto` using a JSON serializer we will trap exceptions and turn them into a `Result`.

```
let fromJson<'a> str =
    try
        let serializer = new DataContractJsonSerializer(typeof<'a>)
        let encoding = System.Text.UTF8Encoding()
        use stream = new System.IO.MemoryStream(encoding.GetBytes(s=str))
        let obj = serializer.ReadObject(stream)
        obj :?> 'a
        |> Result.retn
    with
    | ex ->
        Result.failWithMsg ex.Message
```

The signature of `fromJson` is now `string -> Result<'a>`.

Step 2: Transforming a `TreeDto` into a `Tree`

As before, we transform a `TreeDto` into a `Tree` by recursively looping through the record and its subtrees, turning each one into a `InternalNode` or a `LeafNode`. This time, though, we use `Result` to handle any errors.

```
let rec dtoToTreeOfResults (treeDto:TreeDto<'Leaf,'Node>) :Tree<Result<'Leaf>,Result<'
Node>> =
    let nullLeaf = Unchecked.defaultof<'Leaf>
    let nullNode = Unchecked.defaultof<'Node>

    // check if there is nodeData present
    if treeDto.nodeData <> nullNode then
        if treeDto.subtrees = null then
            LeafNode <| Result.failWithMsg "subtrees must not be null if node data pre
sent"
        else
            let subtrees = treeDto.subtrees |> Array.map dtoToTreeOfResults
            InternalNode (Result.retn treeDto.nodeData,subtrees)
    // check if there is leafData present
    elif treeDto.leafData <> nullLeaf then
        LeafNode <| Result.retn (treeDto.leafData)
    // if both missing then fail
    else
        LeafNode <| Result.failWithMsg "expecting leaf or node data"

// val dtoToTreeOfResults :
//   treeDto:TreeDto<'Leaf,'Node> -> Tree<Result<'Leaf>,Result<'Node>>
```

But uh-oh, we now have a `Tree` where every internal node and leaf is wrapped in a `Result`. It's a tree of `Results`! The actual ugly signature is this: `Tree<Result<'Leaf>,Result<'Node>>`.

But this type is useless as it stands -- what we *really* want is to merge all the errors together and return a `Result` containing a `Tree`.

How can we transform a Tree of Results into a Result of Tree?

The answer is to use a `sequence` function which "swaps" the two types. You can read much more about `sequence` in my series on elevated worlds.

Note that we could also use the slightly more complicated `traverse` variant to combine the `map` and `sequence` into one step, but for the purposes of this demonstration, it's easier to understand if the steps are kept separate.

We need to create our own `sequence` function for the Tree/Result combination. Luckily the creation of a sequence function is a mechanical process:

- For the lower type (`Result`) we need to define `apply` and `return` functions. See here for more details on what `apply` means.
- For the higher type (`Tree`) we need to have a `cata` function, which we do.
- In the catamorphism, each constructor of the higher type (`LeafNode` and `InternalNode` in this case) is replaced by an equivalent that is "lifted" to the `Result` type (e.g. `retn`

`LeafNode <*> data`)

Here is the actual code -- don't worry if you can't understand it immediately. Luckily, we only need to write it once for each combination of types, so for any kind of Tree/Result combination in the future, we're set!

```
/// Convert a tree of Results into a Result of tree
let sequenceTreeOfResult tree =
    // from the lower level
    let (<*>) = Result.apply
    let retn = Result.retn

    // from the traversable level
    let fLeaf data =
        retn LeafNode <*> data

    let fNode data subitems =
        let makeNode data items = InternalNode(data,items)
        let subItems = Result.sequenceSeq subitems
        retn makeNode <*> data <*> subItems

    // do the traverse
    Tree.cata fLeaf fNode tree

// val sequenceTreeOfResult :
//    tree:Tree<Result<'a>,Result<'b>> -> Result<Tree<'a,'b>>
```

Finally, the actual `dtoToTree` function is simple -- just send the `treeDto` through `dtoToTreeOfResults` and then use `sequenceTreeOfResult` to convert the final result into a `Result<Tree<..>>` , which is just what we need.

```
let dtoToTree treeDto =
    treeDto |> dtoToTreeOfResults |> sequenceTreeOfResult

// val dtoToTree : treeDto:TreeDto<'a,'b> -> Result<Tree<'a,'b>>
```

Step 3: Transforming a `GiftDto` into a `Gift`

Again we can use `Tree.map` to convert each leaf and internal node from a DTO to the proper domain type.

But our functions will handle errors, so they need to map a `GiftContentsDto` into a `Result<GiftContents>` and a `GiftDecorationDto` into a `Result<GiftDecoration>`. This results in a Tree of Results again, and so we'll have to use `sequenceTreeOfResult` again to get it back into the correct `Result<Tree<..>>` shape.

Let's start with the helper methods (such as `strToChocolateType`) that convert a string into a proper domain type. This time, they return a `Result` rather than throwing an exception.

```
let strToBookTitle str =
    match str with
    | null -> Result.failWithMsg "BookTitle must not be null"
    | _ -> Result.retn str

let strToChocolateType str =
    match str with
    | "Dark" -> Result.retn Dark
    | "Milk" -> Result.retn Milk
    | "SeventyPercent" -> Result.retn SeventyPercent
    | _ -> Result.failWithMsg (sprintf "ChocolateType %s not recognized" str)

let strToWrappingPaperStyle str =
    match str with
    | "HappyBirthday" -> Result.retn HappyBirthday
    | "HappyHolidays" -> Result.retn HappyHolidays
    | "SolidColor" -> Result.retn SolidColor
    | _ -> Result.failWithMsg (sprintf "WrappingPaperStyle %s not recognized" str)

let strToCardMessage str =
    match str with
    | null -> Result.failWithMsg "CardMessage must not be null"
    | _ -> Result.retn str
```

The case converter methods have to build a `Book` or `Chocolate` from parameters that are `Result`s rather than normal values. This is where lifting functions like `Result.lift2` can help. For details on how this works, see this post on lifting and this one on validation with applicatives.

```
let bookFromDto (dto:GiftContentsDto) =
    let book bookTitle price =
        Book {title=bookTitle; price=price}

    let bookTitle = strToBookTitle dto.bookTitle
    let price = Result.retn dto.price
    Result.lift2 book bookTitle price

let chocolateFromDto (dto:GiftContentsDto) =
    let choc chocType price =
        Chocolate {chocType=chocType; price=price}

    let chocType = strToChocolateType dto.chocolateType
    let price = Result.retn dto.price
    Result.lift2 choc chocType price

let wrappedFromDto (dto:GiftDecorationDto) =
    let wrappingPaperStyle = strToWrappingPaperStyle dto.wrappingPaperStyle
    Result.map Wrapped wrappingPaperStyle

let boxedFromDto (dto:GiftDecorationDto) =
    Result.retn Boxed

let withACardFromDto (dto:GiftDecorationDto) =
    let message = strToCardMessage dto.message
    Result.map WithACard message
```

And finally, the `dtoToGift` function itself is changed to return a `Result` if the `discriminator` is invalid.

As before, this mapping creates a Tree of Results, so we pipe the output of the `Tree.map` through `sequenceTreeOfResult` ...

```
`Tree.map fLeaf fNode giftDto |> sequenceTreeOfResult`
```

... to return a Result of Tree.

Here's the complete code for `dtoToGift` :

```
open TreeDto_WithErrorHandling

/// Transform a GiftDto to a Result<Gift>
let dtoToGift (giftDto:GiftDto) :Result<Gift>=

    let fLeaf (leafDto:GiftContentsDto) =
        match leafDto.discriminator with
        | "Book" -> bookFromDto leafDto
        | "Chocolate" -> chocolateFromDto leafDto
        | _ -> Result.failWithMsg (sprintf "Unknown leaf discriminator '%s'" leafDto.d
iscriminator)

    let fNode (nodeDto:GiftDecorationDto)  =
        match nodeDto.discriminator with
        | "Wrapped" -> wrappedFromDto nodeDto
        | "Boxed" -> boxedFromDto nodeDto
        | "WithACard" -> withACardFromDto nodeDto
        | _ -> Result.failWithMsg (sprintf "Unknown node discriminator '%s'" nodeDto.d
iscriminator)

    // map the tree
    Tree.map fLeaf fNode giftDto |> sequenceTreeOfResult
```

The type signature of `dtoToGift` has changed -- it now returns a `Result<Gift>` rather than just a `Gift`.

```
// val dtoToGift : GiftDto -> Result<GiftUsingTree.Gift>
```

Step 4: Assembling the pipeline

We can now reassemble the pipeline that takes a JSON string and creates a `Gift`.

But changes are needed to work with the new error handling code:

- The `fromJson` function returns a `Result<TreeDto>` but the next function in the pipeline (`dtoToTree`) expects a regular `TreeDto` as input.
- Similarly `dtoToTree` returns a `Result<Tree>` but the next function in the pipeline (`dtoToGift`) expects a regular `Tree` as input.

In both case, `Result.bind` can be used to solve that problem of mis-matched output/input. See here for a more detailed discussion of bind.

Ok, let's try deserializing the `goodJson` string we created earlier.

```
let goodGift = goodJson |> fromJson |> Result.bind dtoToTree |> Result.bind dtoToGift

// check that the description is unchanged
goodGift |> description
// Success "SeventyPercent chocolate in a box wrapped in HappyHolidays paper"
```

That's fine.

Let's see if the error handling has improved now. We'll corrupt the JSON again:

```
let badJson1 = goodJson.Replace("leafData","leafDataXX")

let badJson1_result = badJson1 |> fromJson |> Result.bind dtoToTree |> Result.bind dto
ToGift
// Failure ["The data contract type 'TreeDto' cannot be deserialized because the requi
red data member 'leafData@' was not found."]
```

Great! We get an nice `Failure` case.

Or what if a discriminator is wrong?

```
let badJson2 = goodJson.Replace("Wrapped","Wrapped2")
let badJson2_result = badJson2 |> fromJson |> Result.bind dtoToTree |> Result.bind dto
ToGift
// Failure ["Unknown node discriminator 'Wrapped2'"]
```

or one of the values for the WrappingPaperStyle DU?

```
let badJson3 = goodJson.Replace("HappyHolidays","HappyHolidays2")
let badJson3_result = badJson3 |> fromJson |> Result.bind dtoToTree |> Result.bind dto
ToGift
// Failure ["WrappingPaperStyle HappyHolidays2 not recognized"]
```

Again, nice `Failure` cases.

What's very nice (and this is something that the exception handling approach can't offer) is that if there is more than one error, the various errors can be aggregated so that we get a list of *all* the things that went wrong, rather than just one error at a time.

Let's see this in action by introducing two errors into the JSON string:

```
// create two errors
let badJson4 = goodJson.Replace("HappyHolidays","HappyHolidays2")
                       .Replace("SeventyPercent","SeventyPercent2")
let badJson4_result = badJson4 |> fromJson |> Result.bind dtoToTree |> Result.bind dto
ToGift
// Failure ["WrappingPaperStyle HappyHolidays2 not recognized";
//            "ChocolateType SeventyPercent2 not recognized"]
```

So overall, I'd say that's a success!

The source code for this example is available at this gist.

Summary

We've seen in this series how to define catamorphisms, folds, and in this post in particular, how to use them to solve real world problems. I hope these posts have been useful, and have provided you with some tips and insights that you can apply to your own code.

This series turned out to be a lot longer that I intended, so thanks for making it to the end! Cheers!

In this series of posts, I'll look at how you might handle the common security challenge of authorization. That is, how can you ensure that clients of your code can only do what you want them to do?

This series will sketch out two different approaches, first using an approach called *capability based security*, and second using statically checked types to emulate access tokens.

Interestingly, both approaches tend to produce a cleaner, more modular design as a side effect, which is why I like them!

- A functional approach to authorization. Capability based security and more.
- Constraining capabilities based on identity and role. A functional approach to authorization, part 2.
- Using types as access tokens. A functional approach to authorization, part 3.

A functional approach to authorization

UPDATE: Slides and video from my talk on this topic

In this series of posts, I'll look at how you might handle the common security challenge of authorization. That is, how can you ensure that clients of your code can only do what you want them to do?

This series will sketch out two different approaches, first using an approach called *capability based security*, and second using statically checked types to emulate access tokens.

Interestingly, both approaches tend to produce a cleaner, more modular design as a side effect, which is why I like them!

Before I start, I must mention a major caveat. In a .NET environment, you can generally use reflection to bypass compile-time checking, so the approaches shown here are not about preventing truly malicious attacks so much as helping you create designs that reduce *unintentional* security vulnerabilities.

Finally, I'm no expert on security -- I'm just putting down some of my own thoughts and suggestions. This post is certainly not meant to substitute for a proper full-fledged security design, nor is it a serious study of security practices. If you want to know more, there are links to further reading at the bottom of the post.

Part 1: A configuration example

First, let's start with a simple scenario:

- You have a configuration option that can be set by one part of the the code. Let's say it is a boolean called `DontShowThisMessageAgain`.
- We have a component of the application (the UI say) that wants to set this option.
- In addition, we're also going to assume that the component was written by a malicious developer and is going to try to cause trouble if possible.

So, how should we expose this configuration setting to a potentially malicious caller?

Attempt 1: Give the caller the name of the configuration file

Let's start with a really bad idea. We'll just provide the name of the config file to the caller, and let them change the file themselves.

Here's how this might be written in C# pseudocode:

```
interface IConfiguration
{
    string GetConfigFilename();
}
```

and the caller code would be

```
var filename = config.GetConfigFilename();
// open file
// write new config
// close file
```

Obviously, this is not good! In order for this to work, we have to give the caller the ability to write to any file on the filesystem, and then a malicious caller could delete or corrupt all sorts of things.

You could avoid this to some extent by having strict permissions on the file system, but we're still giving way too much control to the caller.

Attempt 2: Give the caller a TextWriter

Ok, so let's open the file ourselves and just give the caller the opened file stream as a `TextWriter`. That way the caller doesn't need permission to access the file system at all.

But of course, a malicious caller could still corrupt the config file by writing garbage to the file. Again, we're giving way too much control to the caller.

Attempt 3: Give the caller a key/value interface

Let's lock this down by providing the caller an interface that forces them to treat the config file as a key/value store, like this:

```
interface IConfiguration
{
    void SetConfig(string key, string value);
}
```

The caller code is then something like this:

```
config.SetConfig("DontShowThisMessageAgain", "True");
```

That's much better, but because it is a stringly-typed interface, a malicious caller could still corrupt the configuration by setting the value to a non-boolean which would not parse. They could also corrupt all the other configuration keys if they wanted to.

Attempt 4: Give the caller a domain-centric interface

Ok, so rather than having a generic config interface, let's provide an interface that provides specific methods for each configuration setting.

```
enum MessageFlag {
   ShowThisMessageAgain,
   DontShowThisMessageAgain
   }

interface IConfiguration
{
    void SetMessageFlag(MessageFlag value);
    void SetConnectionString(ConnectionString value);
    void SetBackgroundColor(Color value);
}
```

Now the caller can't possibly corrupt the config, because each option is statically typed.

But we still have a problem! What's to stop a malicious caller changing the connection string when they were only supposed to change the message flag?

Attempt 5: Give the caller only the interface they need

Ok, so let's define a new interface that contains *only* the methods the caller should have access to, with all the other methods hidden.

```
interface IWarningMessageConfiguration
{
    void SetMessageFlag(MessageFlag value);
}
```

That's about as locked down as we can get! The caller can *only* do the thing we allow them to do.

In other words, we have just created a design using the Principle Of Least Authority, normally abbreviated to "POLA".

Security as good design

What's interesting about this approach is that it exactly parallels what you would do for good design *anyway*, regardless of a malicious caller.

Here's how I might think about designing this, basing my decisions only on core design principles such information hiding and decoupling.

- If we give the caller a filename, we would be limiting ourselves to file-based config files. By giving the caller a TextWriter, we can make the design more mockable.
- But if we give the caller a TextWriter, we are exposing a specific storage format (XML, JSON, etc) and are also limiting ourselves to text-based storage. By giving the caller a generic KeyValue store, we hide the format and make the implementation choices more flexible.
- But if we give the caller a generic KeyValue store using strings, we are still exposing ourselves to bugs where the value is not a boolean, and we'd have to write validation and tests for that. If we use a statically typed interface instead, we don't have to write any corruption checking code.
- But if we give the caller an interface with too many methods, we are not following the *Interface Segregation Principle*. Hence, we should reduce the number of available methods to the absolute minimum needed by the caller.

Working through a thought process like this, using good design practices only, we end up with exactly the same result as if we had been worried about security!

That is: designing the most minimal interface that the caller needs will both avoid accidental complexity (good design) and increase security (POLA).

Of course, we don't normally have to deal with malicious callers, but we should treat ourselves, as developers, as unintentionally malicious. For example, if there is a extra method in the interface, it might well be used in a different context, which then increases coupling between the two contexts and makes refactoring harder.

So, here's a tip: **design for malicious callers and you will probably end up with more modular code!**

Introducing capability-based security

What we have done above is gradually reduce the surface area to the caller so that by the final design, the caller can only do exactly one thing.

That "one thing" is a "capability". The caller has a capability to set the message flag, and that's all.

"Capability-based" security is a security model that is based on this idea:

- The system provides "capabilities" to clients (in our case, via an implementation of an interface).
- These capabilities encapsulate any access rights that are needed. For example, the very fact that I have access to an implementation of the interface means that I can set that flag. If I did not have permission to set that flag, I would have not been given the

capability (interface) in the first place. (I'll talk more about authorization in the next post).
- Finally, the capabilities can be passed around. For example, I can acquire the capability at startup and then later pass it to the UI layer which can use it as needed.

In other words, we have a "just-in-time" rather than a "just-in-case" model; we pass in the minimal amount of authority as and when needed, rather than having excess "ambient" authority available globally to everyone.

The capability-based model is often focused on operating systems, but it can be mapped to programming languages very nicely, where it is called the object-capability model.

I hope to demonstrate in this post that by using a capability-based approach in your code, you can create better designed and more robust code. In addition, potential security errors will be detectable at *compile-time* rather than at run-time.

As I mentioned above, if your app is trusted, you can always use .NET reflection to "forge" capabilities that you are not entitled to. So, again, the approach shown here is not about preventing truly malicious attacks so much as it about creating a more robust design that reduces *unintentional* security vulnerabilities.

Authority vs. permission

A capability-based security model tends to use the term "authority" rather than "permission". There is a distinction between the two:

- In an *authority* based system, once I have been granted authority to do something, I can pass some or all of that authority to others, add additional constraints of my own, and so on.
- In a *permission* based system, I can ask for permission to do something, but I cannot pass that around to others.

It might seem that an authority based system is more open and "dangerous" than a permission based system. But in a permission based system, if others have access to me and I cooperate with them, I can act as proxy for anything they want to do so, so third-parties can *still* get authority indirectly. Permissions don't really make things more secure -- an attacker just has to use a more convoluted approach.

Here's a concrete example. Let's say Alice trusts me to drive her car, and she is willing to let me borrow it, but she doesn't trust Bob. If I'm friends with Bob, I can let Bob drive the car anyway when Alice is not looking. So if Alice trusts me, she also implicitly trusts anyone that I trust. An authority-based system just makes this explicit. Alice giving me her car keys is giving me the "capability" to drive her car, with full knowledge that I might give the car keys to someone else.

Of course, when I act as a proxy in a permission based system, I can stop cooperating with the third-party if I want to, at which point the third-party loses their access.

The equivalent of that in an authority based system is "revokable authority", which we will see an example of later. In the car key analogy, this might be like having car keys that self-destruct on demand!

Modelling capabilities as functions

An interface with one method can be better realized as a function. So this interface:

```
interface IWarningMessageConfiguration
{
    void SetMessageFlag(MessageFlag value);
}
```

becomes just this function:

```
Action<MessageFlag> messageFlagCapability = // get function;
```

or in F#:

```
let messageFlagCapability = // get function;
```

In a functional approach to capability-based security, each capability is represented by a function rather than an interface.

The nice thing about using functions to represent capabilities is that we can use all the standard functional programming techniques: we can compose them, combine them with combinators, and so on.

The object-capability model vs. the functional programming model

Many of the other requirements of the object-capability model fit well within a functional programming framework. Here is a comparison table:

Object-capability model	Functional programming
No global mutable state is allowed.	No global mutable state is allowed.
Capabilities are always passed around explicitly from parent to child, or from a sender to a receiver.	Functions are values that can be passed as parameters.
Capabilities are never extracted out of the environment ("ambient authority").	Pure functions have all "dependencies" passed in explicitly.
Capabilities cannot be tampered with.	Data is immutable.
Capabilities cannot be forged or cast to other capabilities.	In a uncompromising FP language, there is no reflection or casting available (of course, F# is not strict in this way).
Capabilities should "fail safe". If a capability cannot be obtained, or doesn't work, we must not allow any progress on paths that assumed that it was successful.	In a statically typed language such as F#, we can embed these kinds of control-flow rules into the type system. The use of `Option` is an example of this.

You can see that there is quite a lot of overlap.

One of the *unofficial* goals of the object-capability model is **make security user-friendly by making the security invisible**. I think that this is a great idea, and by passing capabilities as functions, is quite easily achievable.

It's important to note there is one important aspect in which a capability-based model does *not* overlap with a true functional model.

Capabilities are mostly all about (side) effects -- reading or writing the file system, the network, etc. A true functional model would try to wrap them somehow (e.g. in a monad). Personally, using F#, I would generally just allow the side-effects rather than constructing a more complex framework.

But again, as I noted above, the goal of this post is to not to force you into a 100% strict object-capability model, but to borrow some of the same ideas in order to create better designs.

Getting capabilities

A natural question at this point is: where do these capability functions come from?

The answer is, some sort of service that can authorize you to have that capability. In the configuration example, we generally don't do serious authorization, so the configuration service itself will normally provide the capabilities without checking your identity, role or other claims.

But now I need a capability to access the configuration service. Where does that come from? The buck has to stop somewhere!

In OO designs, there is typically a bootstrap/startup stage where all the dependencies are constructed and an IoC container is configured. In a capability based system, a so-called Powerbox plays a similar role of being the starting point for all authority.

Here's the code for a service that provides configuration capabilities:

```
interface IConfigurationCapabilities
{
    Action<MessageFlag> SetMessageFlag();
    Action<ConnectionString> SetConnectionString();
    Action<Color> SetBackgroundColor();
}
```

This code might look very similar to the interface defined earlier, but the difference is that this one will be initialized at startup to return capabilities that are then passed around.

The actual users of the capabilities will not have access to the configuration system at all, just the capabilities they have been given. That is, the capability will be injected into the clients in the same way as a one method interface would be injected in an OO model.

Here's some sample C# pseudocode to demonstrate:

- The capability is obtained at startup.
- The capability is injected into the main window (`ApplicationWindow`) via the constructor.
- The `ApplicationWindow` creates a checkbox.
- The event handler for the checkbox calls the capability.

```
// at startup
var messageFlagCapability =
    configurationCapabilities.SetMessageFlag()
var appWindow = new ApplicationWindow(messageFlagCapability)

// and in the UI class
class ApplicationWindow
{
    // pass in capability in the constructor
    // just as you would an interface
    ApplicationWindow(Action<MessageFlag> messageFlagCapability)
    {
        // set fields
    }

    // setup the check box and register the "OnCheckboxChecked" handler

    // use the capability when the event happens
    void OnCheckboxChecked(CheckBox sender)
    {
        messageFlagCapability(sender.IsChecked)
    }
}
```

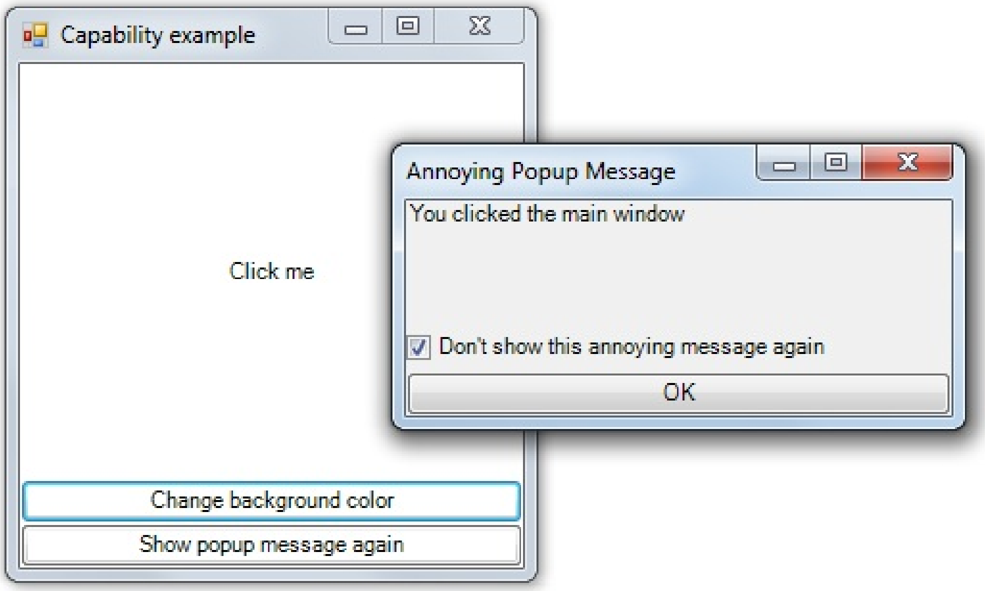
A complete example in F#

Here's the code to a complete example in F# (also available as a gist here).

This example consists of a simple window with a main region and some extra buttons.

- If you click in the main area, an annoying dialog pops up with a "don't show this message again" option.
- One of the buttons allows you to change the background color using the system color picker, and store it in the config.
- The other button allows you to reset the "don't show this message again" option back to false.

It's very crude and very ugly -- no UI designers were hurt in the making of it -- but it should demonstrate the main points so far.



The configuration system

We start with the configuration system. Here's an overview:

- The custom types `MessageFlag`, `ConnectionString`, and `Color` are defined.
- The record type `ConfigurationCapabilities` is defined to hold all the capabilities.
- An in-memory store (`ConfigStore`) is created for the purposes of the demo
- Finally, the `configurationCapabilities` are created using functions that read and write to the `ConfigStore`

```
module Config =

    type MessageFlag  = ShowThisMessageAgain | DontShowThisMessageAgain
    type ConnectionString = ConnectionString of string
    type Color = System.Drawing.Color

    type ConfigurationCapabilities = {
        GetMessageFlag : unit -> MessageFlag
        SetMessageFlag : MessageFlag -> unit
        GetBackgroundColor : unit -> Color
        SetBackgroundColor : Color -> unit
        GetConnectionString  : unit -> ConnectionString
        SetConnectionString : ConnectionString -> unit
        }

    // a private store for demo purposes
    module private ConfigStore =
        let mutable MessageFlag = ShowThisMessageAgain
        let mutable BackgroundColor = Color.White
        let mutable ConnectionString = ConnectionString ""

    // public capabilities
    let configurationCapabilities = {
        GetMessageFlag = fun () -> ConfigStore.MessageFlag
        SetMessageFlag = fun flag -> ConfigStore.MessageFlag <- flag
        GetBackgroundColor = fun () -> ConfigStore.BackgroundColor
        SetBackgroundColor = fun color -> ConfigStore.BackgroundColor <- color
        SetConnectionString = fun _ -> () // ignore
        GetConnectionString = fun () -> ConfigStore.ConnectionString
        SetConnectionString = fun connStr -> ConfigStore.ConnectionString <- connStr
        }
```

The annoying popup dialog

Next, we'll create the annoying popup dialog. This will be triggered whenever you click on the main window, *unless* the "Don't show this message again" option is checked.

The dialog consists of a label control, the message flag checkbox, and the OK button.

Notice that the `createMessageFlagCheckBox` function, which creates the checkbox control, is passed only the two capabilities it needs -- to get and set the flag.

This requires in turn that the main form creation function (`createForm`) is also passed the capabilities. These capabilities, and these capabilities *only* are passed in to the form. The capabilities for setting the background color or connection string are *not* passed in, and thus not available to be (mis)used.

```
module AnnoyingPopupMessage =
    open System.Windows.Forms

    let createLabel() =
        new Label(Text="You clicked the main window", Dock=DockStyle.Top)

    let createMessageFlagCheckBox capabilities  =
        let getFlag,setFlag = capabilities
        let ctrl= new CheckBox(Text="Don't show this annoying message again", Dock=Doc
kStyle.Bottom)
        ctrl.Checked <- getFlag()
        ctrl.CheckedChanged.Add (fun _ -> ctrl.Checked |> setFlag)
        ctrl   // return new control

    let createOkButton (dialog:Form) =
        let ctrl= new Button(Text="OK",Dock=DockStyle.Bottom)
        ctrl.Click.Add (fun _ -> dialog.Close())
        ctrl

    let createForm capabilities =
        let form = new Form(Text="Annoying Popup Message", Width=300, Height=150)
        form.FormBorderStyle <- FormBorderStyle.FixedDialog
        form.StartPosition <- FormStartPosition.CenterParent

        let label = createLabel()
        let messageFlag = createMessageFlagCheckBox capabilities
        let okButton = createOkButton form
        form.Controls.Add label
        form.Controls.Add messageFlag
        form.Controls.Add okButton
        form
```

The main application window

We can now create a main window for our rather silly "application". It consists of:

- A label control that can be clicked to produce the annoying popup (`createClickMeLabel`)
- A button that brings up a color picking dialog to change the background color (`createChangeBackColorButton`)
- A button that resets the message flag to "show" again (`createResetMessageFlagButton`)

All three of these constructor functions are passed capabilities, but capabilities are different in each case.

- The label control is only passed `getFlag` and `setFlag` capabilities
- The color picking dialog is only passed `getColor` and `setColor` capabilities
- The button that resets the message flag is only passed the `setFlag` capability

In the main form (`createMainForm`) the complete set of capabilities are passed in, and they are recombined in various ways as needed for the child controls (`popupMessageCapabilities` , `colorDialogCapabilities`).

In addition, the capability functions are modified:

- A new "SetColor" capability is created from the existing one, with the addition of changing the form's background as well.
- The flag capabilities are converted from the domain type (`MessageFlag`) to bools that can be used directly with the checkbox.

Here's the code:

```
module UserInterface =
    open System.Windows.Forms
    open System.Drawing

    let showPopupMessage capabilities owner =
        let getFlag,setFlag = capabilities
        let popupMessage = AnnoyingPopupMessage.createForm (getFlag,setFlag)
        popupMessage.Owner <- owner
        popupMessage.ShowDialog() |> ignore // don't care about result

    let showColorDialog capabilities owner =
        let getColor,setColor = capabilities
        let dlg = new ColorDialog(Color=getColor())
        let result = dlg.ShowDialog(owner)
        if result = DialogResult.OK then
            dlg.Color |> setColor

    let createClickMeLabel capabilities owner =
        let getFlag,_ = capabilities
        let ctrl= new Label(Text="Click me", Dock=DockStyle.Fill, TextAlign=ContentAli
gnment.MiddleCenter)
        ctrl.Click.Add (fun _ ->
            if getFlag() then showPopupMessage capabilities owner)
        ctrl  // return new control

    let createChangeBackColorButton capabilities owner =
        let ctrl= new Button(Text="Change background color", Dock=DockStyle.Bottom)
        ctrl.Click.Add (fun _ -> showColorDialog capabilities owner)
        ctrl

    let createResetMessageFlagButton capabilities =
        let setFlag = capabilities
        let ctrl= new Button(Text="Show popup message again", Dock=DockStyle.Bottom)
        ctrl.Click.Add (fun _ -> setFlag Config.ShowThisMessageAgain)
        ctrl

    let createMainForm capabilities =
        // get the individual component capabilities from the parameter
```

```
        let getFlag,setFlag,getColor,setColor = capabilities

        let form = new Form(Text="Capability example", Width=500, Height=300)
        form.BackColor <- getColor() // update the form from the config

        // transform color capability to change form as well
        let newSetColor color =
            setColor color          // change config
            form.BackColor <- color  // change form as well

        // transform flag capabilities from domain type to bool
        let getBoolFlag() =
            getFlag() = Config.ShowThisMessageAgain
        let setBoolFlag bool =
            if bool
            then setFlag Config.ShowThisMessageAgain
            else setFlag Config.DontShowThisMessageAgain

        // set up capabilities for child objects
        let colorDialogCapabilities = getColor,newSetColor
        let popupMessageCapabilities = getBoolFlag,setBoolFlag

        // setup controls with their different capabilities
        let clickMeLabel = createClickMeLabel popupMessageCapabilities form
        let changeColorButton = createChangeBackColorButton colorDialogCapabilities fo
rm
        let resetFlagButton = createResetMessageFlagButton setFlag

        // add controls
        form.Controls.Add clickMeLabel
        form.Controls.Add changeColorButton
        form.Controls.Add resetFlagButton

        form  // return form
```

The startup code

Finally, the top-level module, here called `Startup`, gets some of the capabilities from the Configuration subsystem, and combines them into a tuple that can be passed to the main form. The `ConnectionString` capabilities are *not* passed in though, so there is no way the form can accidentally show it to a user or update it.

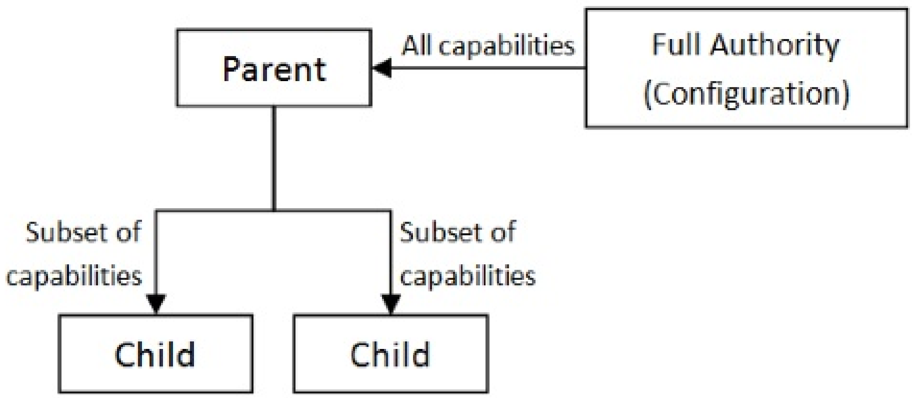
```
module Startup =

    // set up capabilities
    let configCapabilities = Config.configurationCapabilities
    let formCapabilities =
        configCapabilities.GetMessageFlag,
        configCapabilities.SetMessageFlag,
        configCapabilities.GetBackgroundColor,
        configCapabilities.SetBackgroundColor

    // start
    let form = UserInterface.createMainForm formCapabilities
    form.ShowDialog() |> ignore
```

Summary of Part 1

As you can see, this code is very similar to an OO system designed with dependency injection. There is no global access to capabilities, only those passed in from the parent.



Of course, the use of functions to parameterize behavior like this is nothing special. It's one of the most fundamental functional programming techniques. So this code is not really showing any new ideas, rather it is just demonstrating how a standard functional programming approach can be applied to enforce access paths.

Some common questions at this point:

Question: This seems like extra work. Why do I need to do this at all?

If you have a simple system, you certainly don't need to do this. But here's where it might be useful:

- You have a system which uses fine-grained authorization already, and you want to make this more explicit and easier to use in practice.
- You have a system which runs at a high privilege but has strict requirements about leaking data or performing actions in an unauthorized context.

In these situations, I believe that is very important to be *explicit* about what the capabilities are at *all points* in the codebase, not just in the UI layer. This not only helps with compliance and auditing needs, but also has the practical benefit that it makes the code more modular and easier to maintain.

Question: What's the difference between this approach and dependency injection?

Dependency injection and a capability-based model have different goals. Dependency injection is all about decoupling, while capabilities are all about controlling access. As we have seen, both approaches end up promoting similar designs.

Question: What happens if I have hundreds of capabilities that I need to pass around?

It seems like this should be a problem, but in practice it tends not to be. For one thing, judicious use of partial application means that capabilities can be baked in to a function before passing it around, so that child objects are not even aware of them.

Secondly, it is very easy -- just a few lines -- to create simple record types that contain a group of capabilities (as I did with the `ConfigurationCapabilities` type) and pass those around if needed.

Question: What's to stop someone accessing global capabilities without following this approach?

Nothing in C# or F# can stop you accessing global public functions. Just like other best practices, such as avoiding global variables, we have to rely on self-discipline (and maybe code reviews) to keep us on the straight and narrow path!

But in the third part of this series, we'll look at a way to prevent access to global functions by using access tokens.

Question: Aren't these just standard functional programming techniques?

Yes. I'm not claiming to be doing anything clever here!

Question: These capability functions have side-effects. What's up with that?

Yes, these capability functions are not pure. The goal here is not about being pure -- it's about being explicit about the provision of capabilities.

Even if we used a pure `IO` context (e.g. in Haskell) it would not help control access to capabilities. That is, in the context of security, there's a big difference between the capability to change a password or credit card vs. the capability to change a background color configuration, even though they are both just "IO" from a computation point of view.

Creating pure capabilities is possible but not very easy to do in F#, so I'm going to keep it out of scope for this post.

Question: What's your response to what (some person) wrote? And why didn't you cite (some paper)?

This is a blog post, not an academic paper. I'm not an expert in this area, but just doing some experiments of my own.

More importantly, as I said earlier, my goal here is very different from security experts -- I'm *not* attempting to develop a ideal security model. Rather, I'm just trying to encourage some *good design* practices that can help pragmatic developers avoid accidental vulnerabilities in their code.

I've got more questions...

Some additional questions are answered at the end of part 2, so read those answers first. Otherwise please add your question in the comments below, and I'll try to address it.

Further reading

The ideas on capability-based security here are mostly derived from the work of Mark Miller and Marc Stiegler, and the erights.org website, although my version is cruder and simpler.

For a more complete understanding, I suggest you follow up on the links below:

- The Wikipedia articles on Capability-based security and Object-capability model are a good starting point.
- What is a Capability, Anyway? by Jonathan Shapiro of the EROS project. He also discusses ACL-based security vs. a capability-based model.
- "The Lazy Programmer's Guide to Secure Computing", a great video on capability-based security by Marc Stiegler. Don't miss the last 5 mins (starting around 1h:02m:10s)!
- "Object Capabilities for Security", a good talk by David Wagner.

A lot of work has been done on hardening languages for security and safety. For example the E Language and Mark Miller's thesis on the E Language(PDF); the Joe-E Language built on top of Java; Google's Caja built over JavaScript; Emily, a capability based language derived from OCaml; and Safe Haskell(PDF).

My approach is not about strict safeness so much as proactively designing to avoid unintentional breaches, and the references above do not focus on very much on design specifically. The most useful thing I have found is a section on capability patterns in E.

Also, if you like this kind of thing, then head over to LtU where there are a number of discussions, such as this one and this one and this paper.

Coming up next

In the next post, we'll look at how to constrain capabilities based on claims such as the current user's identity and role.

NOTE: All the code for this post is available as a gist here.

Constraining capabilities based on identity and role

UPDATE: Slides and video from my talk on this topic

In the previous post, we started looking at "capabilities" as the basis for ensuring that code could not do any more than it was supposed to do. And I demonstrated this with a simple application that changed a configuration flag.

In this post, we'll look at how to constrain capabilities based on the current user's identity and role.

So let's switch from the configuration example to a typical situation where stricter authorization is required.

Database capabilities example

Consider a website and call-centre with a backing database. We have the following security rules:

- A customer can only view or update their own record in the database (via the website)
- A call-centre operator can view or update any record in the database

This means that at some point, we'll have to do some authorization based on the identity and role of the user. (We'll assume that the user has been authenticated successfully).

The tendency in many web frameworks is to put the authorization in the UI layer, often in the controller. My concern about this approach is that once you are "inside" (past the gateway), any part of the app has full authority to access the database, and it is all to easy for code to do the wrong thing by mistake, resulting in a security breach.

Not only that, but because the authority is everywhere ("ambient"), it is hard to review the code for potential security issues.

To avoid these issues, let's instead put the access logic as "low" as possible, in the database access layer in this case.

We'll start with an obvious approach. We'll add the identity and role to each database call and then do authorization there.

The following method assumes that there is a `CustomerIdBelongsToPrincipal` function that checks whether the customer id being accessed is owned by the principal requesting access. Then, if the `customerId` does belong to the principal, or the principal has the role of "CustomerAgent", the access is granted.

```
public class CustomerDatabase
{
    public CustomerData GetCustomer(CustomerId id, IPrincipal principal)
    {
        if ( CustomerIdBelongsToPrincipal(id, principal) ||
             principal.IsInRole("CustomerAgent") )
        {
            // get customer data
        }
        else
        {
            // throw authorization exception
        }
    }
}
```

Note that I have deliberately added the `IPrincipal` to the method signature -- we are not allowing any "magic" where the principal is fetched from a global context. As with the use of any global, having implicit access hides the dependencies and makes it hard to test in isolation.

Here's the F# equivalent, using a Success/Failure return value rather than throwing exceptions:

```
let getCustomer id principal =
    if customerIdBelongsToPrincipal id principal ||
       principal.IsInRole("CustomerAgent")
    then
        // get customer data
        Success "CustomerData"
    else
        Failure AuthorizationFailed
```

This "inline" authorization approach is all too common, but unfortunately it has many problems.

- It mixes up security concerns with the database logic. If the authorization logic gets more complicated, the code will also get more complicated.
- It throws an exception (C#) or returns an error (F#) if the authorization fails. It would be nice if we could tell *in advance* if we had the authorization rather than waiting until the last minute.

Let's compare this with a capability-based approach. Instead of directly getting a customer, we first obtain the *capability* of doing it.

```
class CustomerDatabase
{
    // "real" code is hidden from public view
    private CustomerData GetCustomer(CustomerId id)
    {
        // get customer data
    }

    // Get the capability to call GetCustomer
    public Func<CustomerId,CustomerData> GetCustomerCapability(CustomerId id, IPrincip
al principal)
    {
        if ( CustomerIdBelongsToPrincipal(id, principal) ||
             principal.IsInRole("CustomerAgent") )
        {
            // return the capability (the real method)
            return GetCustomer;
        }
        else
        {
            // throw authorization exception
        }
    }
}
```

As you can see, if the authorization succeeds, a reference to the `GetCustomer` method is returned to the caller.

It might not be obvious, but the code above has a rather large security hole. I can request the capability for a particular customer id, but I get back a function that can called for *any* customer id! That's not very safe, is it?

What we need to is "bake in" the customer id to the capability, so that it can't be misused. The return value will now be a `Func<CustomerData>`, with the customer id not available to be passed in any more.

```
class CustomerDatabase
{
    // "real" code is hidden from public view
    private CustomerData GetCustomer(CustomerId id)
    {
        // get customer data
    }

    // Get the capability to call GetCustomer
    public Func<CustomerData> GetCustomerCapability(CustomerId id, IPrincipal principal
)
    {
        if ( CustomerIdBelongsToPrincipal(id, principal) ||
             principal.IsInRole("CustomerAgent") )
        {
            // return the capability (the real method)
            return ( () => GetCustomer(id) );
        }
        else
        {
            // throw authorization exception
        }
    }
}
```

With this separation of concerns in place, we can now handle failure nicely, by returning an *optional* value which is present if we get the capability, or absent if not. That is, we know whether we have the capability *at the time of trying to obtain it*, not later on when we try to use it.

```
class CustomerDatabase
{
    // "real" code is hidden from public view
    // and doesn't need any checking of identity or role
    private CustomerData GetCustomer(CustomerId id)
    {
        // get customer data
    }

    // Get the capability to call GetCustomer. If not allowed, return None.
    public Option<Func<CustomerData>> GetCustomerCapability(CustomerId id, IPrincipal
principal)
    {
        if (CustomerIdBelongsToPrincipal(id, principal) ||
             principal.IsInRole("CustomerAgent"))
        {
            // return the capability (the real method)
            return Option<Func<CustomerData>>.Some( () => GetCustomer(id) );
        }
        else
        {
            return Option<Func<CustomerData>>.None();
        }
    }
}
```

This assumes that we're using some sort of `Option` type in C# rather than just returning null!

Finally, we can put the authorization logic into its own class (say `CustomerDatabaseCapabilityProvider`), to keep the authorization concerns separate from the `CustomerDatabase` .

We'll have to find some way of keeping the "real" database functions private to all other callers though. For now, I'll just assume the database code is in a different assembly, and mark the code `internal` .

```
// not accessible to the business layer
internal class CustomerDatabase
{
    // "real" code is hidden from public view
    private CustomerData GetCustomer(CustomerId id)
    {
        // get customer data
    }
}

// accessible to the business layer
public class CustomerDatabaseCapabilityProvider
{
    CustomerDatabase _customerDatabase;

    // Get the capability to call GetCustomer
    public Option<Func<CustomerData>> GetCustomerCapability(CustomerId id, IPrincipal
principal)
    {
        if (CustomerIdBelongsToPrincipal(id, principal) ||
             principal.IsInRole("CustomerAgent"))
        {
            // return the capability (the real method)
            return Option<Func<CustomerData>>.Some( () => _customerDatabase.GetCustome
r(id) );
        }
        else
        {
            return Option<Func<CustomerData>>.None();
        }
    }
}
```

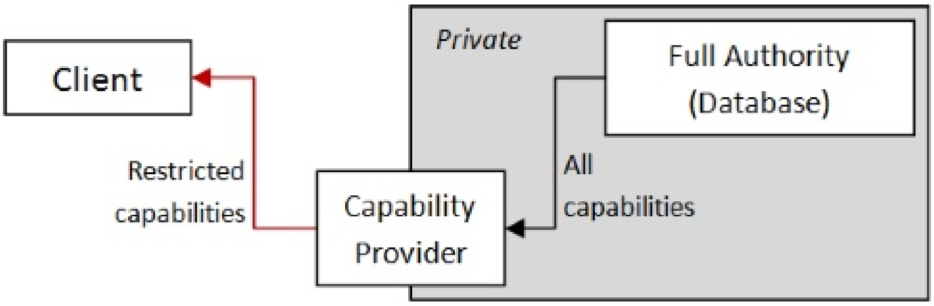
And here's the F# version of the same code:

```
/// not accessible to the business layer
module internal CustomerDatabase =
    let getCustomer (id:CustomerId) :CustomerData =
        // get customer data

/// accessible to the business layer
module CustomerDatabaseCapabilityProvider =

    // Get the capability to call getCustomer
    let getCustomerCapability (id:CustomerId) (principal:IPrincipal) =
        let principalId = GetIdForPrincipal(principal)
        if (principalId = id) || principal.IsInRole("CustomerAgent") then
            Some ( fun () -> CustomerDatabase.getCustomer id )
        else
            None
```

Here's a diagram that represents this design:



Problems with this model

In this model, the caller is isolated from the `CustomerDatabase`, and the `CustomerDatabaseCapabilityProvider` acts as a proxy between them.

Which means, as currently designed, for every function available in `CustomerDatabase` there must be a parallel function available in `CustomerDatabaseCapabilityProvider` as well. We can see that this approach will not scale well.

It would be nice if we had a way to generally get capabilities for a *whole set* of database functions rather than one at a time. Let's see if we can do that!

Restricting and transforming capabilities

The `getCustomer` function in `CustomerDatabase` can be thought of as a capability with no restrictions, while the `getCustomerCapability` returns a capability restricted by identity and role.

But note that the two function signatures are similar (`CustomerId -> CustomerData` vs `unit -> CustomerData`), and so they are almost interchangeable from the callers point of view. In a sense, then, the second capability is a transformed version of the first, with additional restrictions.

Transforming functions to new functions! This is something we can easily do.

So, let's write a transformer that, given *any* function of type `CustomerId -> 'a`, we return a function with the customer id baked in (`unit -> 'a`), but only if the authorization requirements are met.

```
module CustomerCapabilityFilter =

    // Get the capability to use any function that has a CustomerId parameter
    // but only if the caller has the same customer id or is a member of the
    // CustomerAgent role.
    let onlyForSameIdOrAgents (id:CustomerId) (principal:IPrincipal) (f:CustomerId ->
'a) =
        let principalId = GetIdForPrincipal(principal)
        if (principalId = id) || principal.IsInRole("CustomerAgent") then
            Some (fun () -> f id)
        else
            None
```

The type signature for the `onlyForSameIdOrAgents` function is `(CustomerId -> 'a) -> (unit -> 'a) option`. It accepts any `CustomerId` based function and returns, maybe, the same function *with the customer id already applied* if the authorization succeeds. If the authorization does not succeed, `None` is returned instead.

You can see that this function will work generically with *any* function that has a `CustomerId` as the first parameter. That could be "get", "update", "delete", etc.

So for example, given:

```
module internal CustomerDatabase =
    let getCustomer (id:CustomerId) =
        // get customer data
    let updateCustomer (id:CustomerId) (data:CustomerData) =
        // update customer data
```

We can create restricted versions now, for example at the top level bootstrapper or controller:

```
let principal = // from context
let id = // from context

// attempt to get the capabilities
let getCustomerOnlyForSameIdOrAgents =
    onlyForSameIdOrAgents id principal CustomerDatabase.getCustomer

let updateCustomerOnlyForSameIdOrAgents =
    onlyForSameIdOrAgents id principal CustomerDatabase.updateCustomer
```

The types of `getCustomerOnlyForSameIdOrAgents` and `updateCustomerOnlyForSameIdOrAgents` are similar to the original functions in the database module, but with `CustomerId` replaced with `unit`:

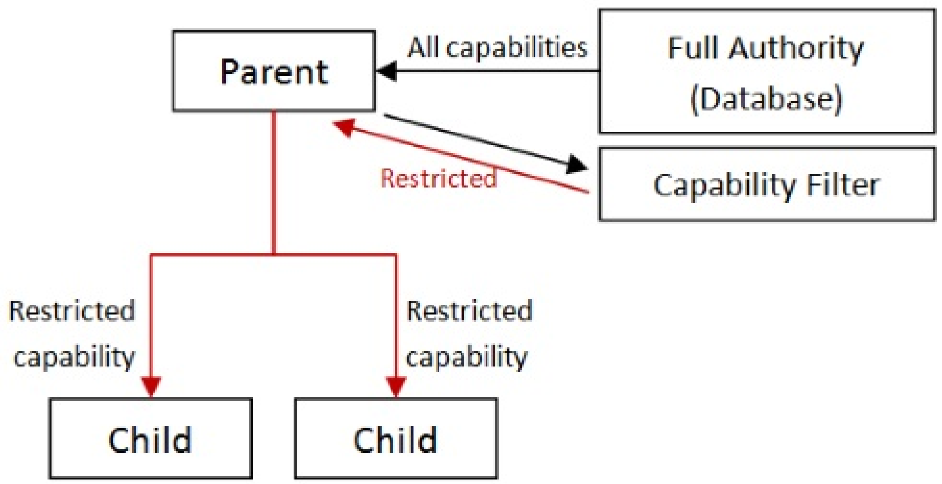
```
val getCustomerOnlyForSameIdOrAgents :
      (unit -> CustomerData) option
val updateCustomerOnlyForSameIdOrAgents :
      (unit -> CustomerData -> unit) option
```

The `updateCustomerOnlyForSameIdOrAgents` has a extra `CustomerData` parameter, so the extra unit where the `CustomerId` used to be is a bit ugly. If this is too annoying, you could easily create other versions of the function which handle this more elegantly. I'll leave that as an exercise for the reader!

So now we have an option value that might or might not contain the capability we wanted. If it does, we can create a child component and pass in the capability. If it does not, we can return some sort of error, or hide a element from a view, depending on the type of application.

```
match getCustomerOnlyForSameIdOrAgents with
| Some cap -> // create child component and pass in the capability
| None ->     // return error saying that you don't have the capability to get the data
```

Here's a diagram that represents this design:



More transforms on capabilities

Because capabilities are functions, we can easily create new capabilities by chaining or combining transformations.

For example, we could create a separate filter function for each business rule, like this:

```
module CustomerCapabilityFilter =

    let onlyForSameId (id:CustomerId) (principal:IPrincipal) (f:CustomerId -> 'a) =
        if customerIdBelongsToPrincipal id principal then
            Some (fun () -> f id)
        else
            None

    let onlyForAgents (id:CustomerId) (principal:IPrincipal) (f:CustomerId -> 'a) =
        if principal.IsInRole("CustomerAgent") then
            Some (fun () -> f id)
        else
            None
```

For the first business rule, `onlyForSameId`, we return a capability with the customer id baked in, as before.

The second business rule, `onlyForAgents`, doesn't mention customer ids anywhere, so why do we restrict the function parameter to `CustomerId -> 'a`? The reason is that it enforces that this rule *only* applies to customer centric capabilities, not ones relating to products or payments, say.

But now, to make the output of this filter compatible with the first rule (`unit -> 'a`), we need to pass in a customer id and partially apply it too. It's a bit of a hack but it will do for now.

We can also write a generic combinator that returns the first valid capability from a list.

```
// given a list of capability options,
// return the first good one, if any
let first capabilityList =
    capabilityList |> List.tryPick id
```

It's a trivial implementation really -- this is the kind of helper function that is just to help the code be a little more self-documenting.

With this in place, we can apply the rules separately, take the two filters and combine them into one.

```
let getCustomerOnlyForSameIdOrAgents =
    let f = CustomerDatabase.getCustomer
    let cap1 = onlyForSameId id principal f
    let cap2 = onlyForAgents id principal f
    first [cap1; cap2]
// val getCustomerOnlyForSameIdOrAgents : (CustomerId -> CustomerData) option
```

Or let's say we have some sort of restriction; the operation can only be performed during business hours, say.

```
let onlyIfDuringBusinessHours (time:DateTime) f =
    if time.Hour >= 8 && time.Hour <= 17 then
        Some f
    else
        None
```

We can write another combinator that restricts the original capability. This is just a version of "bind".

```
// given a capability option, restrict it
let restrict filter originalCap =
    originalCap
    |> Option.bind filter
```

With this in place, we can restrict the "agentsOnly" capability to business hours:

```
let getCustomerOnlyForAgentsInBusinessHours =
    let f = CustomerDatabase.getCustomer
    let cap1 = onlyForAgents id principal f
    let restriction f = onlyIfDuringBusinessHours (DateTime.Now) f
    cap1 |> restrict restriction
```

So now we have created a new capability, "Customer agents can only access customer data during business hours", which tightens the data access logic a bit more.

We can combine this with the previous `onlyForSameId` filter to build a compound capability which can access customer data:

- if you have the same customer id (at any time of day)
- if you are a customer agent (only during business hours)

```
let getCustomerOnlyForSameId =
    let f = CustomerDatabase.getCustomer
    onlyForSameId id principal f

let getCustomerOnlyForSameId_OrForAgentsInBusinessHours =
    let cap1 = getCustomerOnlyForSameId
    let cap2 = getCustomerOnlyForAgentsInBusinessHours
    first [cap1; cap2]
```

As you can see, this approach is a useful way to build complex capabilities from simpler ones.

Additional transforms

It should be obvious that you can easily create additional transforms which can extend capabilities in other ways. Some examples:

- a capability that writes to an audit log on each execution.
- a capability that can only be performed once.
- a capability that can be revoked when needed.
- a capability that is throttled and can only be performed a limited number of times in a given time period (such as password change attempts).

And so on.

Here are implementations of the first three of them:

```
/// Uses of the capability will be audited
let auditable capabilityName f =
    fun x ->
        // simple audit log!
        printfn "AUDIT: calling %s with %A" capabilityName  x
        // use the capability
        f x

/// Allow the function to be called once only
let onlyOnce f =
    let allow = ref true
    fun x ->
        if !allow then   //! is dereferencing not negation!
            allow := false
            f x
        else
            Failure OnlyAllowedOnce

/// Return a pair of functions: the revokable capability,
/// and the revoker function
let revokable f =
    let allow = ref true
    let capability = fun x ->
        if !allow then  //! is dereferencing not negation!
            f x
        else
            Failure Revoked
    let revoker() =
        allow := false
    capability, revoker
```

Let's say that we have an `updatePassword` function, such as this:

```
module internal CustomerDatabase =
    let updatePassword (id,password) =
        Success "OK"
```

We can then create a auditable version of `updatePassword` :

```
let updatePasswordWithAudit x =
    auditable "updatePassword" CustomerDatabase.updatePassword x
```

And then test it:

```
updatePasswordWithAudit (1,"password")
updatePasswordWithAudit (1,"new password")
```

The results are:

```
AUDIT: calling updatePassword with (1, "password")
AUDIT: calling updatePassword with (1, "new password")
```

Or, we could create a one-time only version:

```
let updatePasswordOnce =
    onlyOnce CustomerDatabase.updatePassword
```

And then test it:

```
updatePasswordOnce (1,"password") |> printfn "Result 1st time: %A"
updatePasswordOnce (1,"password") |> printfn "Result 2nd time: %A"
```

The results are:

```
Result 1st time: Success "OK"
Result 2nd time: Failure OnlyAllowedOnce
```

Finally, we can create a revokable function:

```
let revokableUpdatePassword, revoker =
    revokable CustomerDatabase.updatePassword
```

And then test it:

```
revokableUpdatePassword (1,"password") |> printfn "Result 1st time before revoking: %A"

revokableUpdatePassword (1,"password") |> printfn "Result 2nd time before revoking: %A"

revoker()
revokableUpdatePassword (1,"password") |> printfn "Result 3nd time after revoking: %A"
```

With the following results:

```
Result 1st time before revoking: Success "OK"
Result 2nd time before revoking: Success "OK"
Result 3nd time after revoking: Failure Revoked
```

The code for all these F# examples is available as a gist here.

A complete example in F#

Here's the code to a complete application in F# (also available as a gist here).

This example consists of a simple console app that allows you to get and update customer records.

- The first step is to login as a user. "Alice" and "Bob" are normal users, while "Zelda" has a customer agent role.
- Once logged in, you can select a customer to edit. Again, you are limited to a choice between "Alice" and "Bob". (I'm sure you can hardly contain your excitement)
- Once a customer is selected, you are presented with some (or none) of the following options:
    - Get a customer's data.
    - Update a customer's data.
    - Update a customer's password.

Which options are shown depend on which capabilities you have. These in turn are based on who you are logged in as, and which customer is selected.

Implementing the domain

We'll start with the core domain types that are shared across the application:

```
module Domain =
    open Rop

    type CustomerId = CustomerId of int
    type CustomerData = CustomerData of string
    type Password = Password of string

    type FailureCase =
        | AuthenticationFailed of string
        | AuthorizationFailed
        | CustomerNameNotFound of string
        | CustomerIdNotFound of CustomerId
        | OnlyAllowedOnce
        | CapabilityRevoked
```

The `FailureCase` type documents all possible things that can go wrong at the top-level of the application. See the "Railway Oriented Programming" talk for more discussion on this.

Defining the capabilities

Next, we document all the capabilities that are available in the application. To add clarity to the code, each capability is given a name (i.e. a type alias).

```
type GetCustomerCap = unit -> SuccessFailure<CustomerData,FailureCase>
```

Finally, the `CapabilityProvider` is a record of functions, each of which accepts a customer id and principal, and returns an optional capability of the specified type. This record is created in the top level model and then passed around to the child components.

Here's the complete code for this module:

```
module Capabilities =
    open Rop
    open Domain

    // capabilities
    type GetCustomerCap = unit -> SuccessFailure<CustomerData,FailureCase>
    type UpdateCustomerCap = unit -> CustomerData -> SuccessFailure<unit,FailureCase>
    type UpdatePasswordCap = Password -> SuccessFailure<unit,FailureCase>

    type CapabilityProvider = {
        /// given a customerId and IPrincipal, attempt to get the GetCustomer capabili
ty
        getCustomer : CustomerId -> IPrincipal -> GetCustomerCap option
        /// given a customerId and IPrincipal, attempt to get the UpdateCustomer capab
ility
        updateCustomer : CustomerId -> IPrincipal -> UpdateCustomerCap option
        /// given a customerId and IPrincipal, attempt to get the UpdatePassword capab
ility
        updatePassword : CustomerId -> IPrincipal -> UpdatePasswordCap option
        }
```

This module references a `SuccessFailure` result type similar to the one discussed here, but which I won't show.

Implementing authentication

Next, we'll roll our own little authentication system. Note that when the user "Zelda" is authenticated, the role is set to "CustomerAgent".

```
module Authentication =
    open Rop
    open Domain

    let customerRole = "Customer"
    let customerAgentRole = "CustomerAgent"

    let makePrincipal name role =
        let iden = GenericIdentity(name)
        let principal = GenericPrincipal(iden,[|role|])
        principal :> IPrincipal

    let authenticate name =
        match name with
        | "Alice" | "Bob" ->
            makePrincipal name customerRole  |> Success
        | "Zelda" ->
            makePrincipal name customerAgentRole |> Success
        | _ ->
            AuthenticationFailed name |> Failure

    let customerIdForName name =
        match name with
        | "Alice" -> CustomerId 1 |> Success
        | "Bob" -> CustomerId 2 |> Success
        | _ -> CustomerNameNotFound name |> Failure

    let customerIdOwnedByPrincipal customerId (principle:IPrincipal) =
        principle.Identity.Name
        |> customerIdForName
        |> Rop.map (fun principalId -> principalId = customerId)
        |> Rop.orElse false
```

The `customerIdForName` function attempts to find the customer id associated with a particular name, while the `customerIdOwnedByPrincipal` compares this id with another one.

Implementing authorization

Here are the functions related to authorization, very similar to what was discussed above.

```
module Authorization =
    open Rop
    open Domain

    let onlyForSameId (id:CustomerId) (principal:IPrincipal) (f:CustomerId -> 'a) =
        if Authentication.customerIdOwnedByPrincipal id principal then
            Some (fun () -> f id)
        else
            None
```

```
    let onlyForAgents (id:CustomerId) (principal:IPrincipal) (f:CustomerId -> 'a) =
        if principal.IsInRole(Authentication.customerAgentRole) then
            Some (fun () -> f id)
        else
            None

    let onlyIfDuringBusinessHours (time:DateTime) f =
        if time.Hour >= 8 && time.Hour <= 17 then
            Some f
        else
            None

    // constrain who can call a password update function
    let passwordUpdate (id:CustomerId) (principal:IPrincipal) (f:CustomerId*Password -
> 'a) =
        if Authentication.customerIdOwnedByPrincipal id principal then
            Some (fun password -> f (id,password))
        else
            None

    // return the first good capability, if any
    let first capabilityList =
        capabilityList |> List.tryPick id

    // given a capability option, restrict it
    let restrict filter originalCap =
        originalCap
        |> Option.bind filter

    /// Uses of the capability will be audited
    let auditable capabilityName principalName f =
        fun x ->
            // simple audit log!
            let timestamp = DateTime.UtcNow.ToString("u")
            printfn "AUDIT: User %s used capability %s at %s" principalName capability
Name timestamp
            // use the capability
            f x

    /// Return a pair of functions: the revokable capability,
    /// and the revoker function
    let revokable f =
        let allow = ref true
        let capability = fun x ->
            if !allow then  //! is dereferencing not negation!
                f x
            else
                Failure CapabilityRevoked
        let revoker() =
            allow := false
        capability, revoker
```

Implementing the database

The functions related to database access are similar to those in the earlier examples, only this time we have implemented a crude in-memory database (just a `Dictionary`).

```
module CustomerDatabase =
    open Rop
    open System.Collections.Generic
    open Domain

    let private db = Dictionary<CustomerId,CustomerData>()

    let getCustomer id =
        match db.TryGetValue id with
        | true, value -> Success value
        | false, _ -> Failure (CustomerIdNotFound id)

    let updateCustomer id data =
        db.[id] <- data
        Success ()

    let updatePassword (id:CustomerId,password:Password) =
        Success ()   // dummy implementation
```

Implementing the business services

Next we have the "business services" (for lack of better word) where all the work gets done.

```
module BusinessServices =
    open Rop
    open Domain

    // use the getCustomer capability
    let getCustomer capability =
        match capability() with
        | Success data -> printfn "%A" data
        | Failure err -> printfn ".. %A" err

    // use the updateCustomer capability
    let updateCustomer capability =
        printfn "Enter new data: "
        let customerData = Console.ReadLine() |> CustomerData
        match capability () customerData  with
        | Success _ -> printfn "Data updated"
        | Failure err -> printfn ".. %A" err

    // use the updatePassword capability
    let updatePassword capability =
        printfn "Enter new password: "
        let password = Console.ReadLine() |> Password
        match capability password  with
        | Success _ -> printfn "Password updated"
        | Failure err -> printfn ".. %A" err
```

Note that each of these functions is passed in only the capability needed to do its job. This code knows nothing about databases, or anything else.

Yes, in this crude example, the code is reading and writing directly to the console. Obviously in a more complex (and less crude!) design, the inputs to these functions would be passed in as parameters.

Here's a simple exercise: replace the direct access to the console with a capability such as `getDataWithPrompt`?

Implementing the user interface

Now for the user interface module, where most of the complex code lies.

First up is a type (`CurrentState`) that represents the state of the user interface.

- When we're `LoggedOut` there is no `IPrincipal` available.
- When we're `LoggedIn` there is a `IPrincipal` available, but no selected customer.
- When we're in the `CustomerSelected` state there is both a `IPrincipal` and a `CustomerId` available.
- Finally, the `Exit` state is a signal to the app to shutdown.

I very much like using a "state" design like this, because it ensures that we can't accidentally access data that we shouldn't. For example, we literally cannot access a customer when none is selected, because there is no customer id in that state!

For each state, there is a corresponding function.

`loggedOutActions` is run when we are in the `LoggedOut` state. It presents the available actions to you, and changes the state accordingly. You can log in as a user, or exit. If the login is successful (`authenticate name` worked) then the state is changed to `LoggedIn` .

`loggedInActions` is run when we are in the `LoggedIn` state. You can select a customer, or log out. If the customer selection is successful (`customerIdForName customerName` worked) then the state is changed to `CustomerSelected` .

`selectedCustomerActions` is run when we are in the `CustomerSelected` state. This works as follows:

- First, find out what capabilities we have.
- Next convert each capability into a corresponding menu text (using `Option.map` because the capability might be missing), then remove the ones that are None.
- Next, read a line from input, and depending on what it is, call one of the "business services" (`getCustomer` , `updateCustomer` , or `updatePassword`).

Finally the `mainUiLoop` function loops around until the state is set to `Exit` .

```
module UserInterface =
    open Rop
    open Domain
    open Capabilities

    type CurrentState =
        | LoggedOut
        | LoggedIn of IPrincipal
        | CustomerSelected of IPrincipal * CustomerId
        | Exit

    /// do the actions available while you are logged out. Return the new state
    let loggedOutActions originalState =
        printfn "[Login] enter Alice, Bob, Zelda, or Exit: "
        let action = Console.ReadLine()
        match action with
        | "Exit"  ->
            // Change state to Exit
            Exit
        | name ->
            // otherwise try to authenticate the name
            match Authentication.authenticate name with
            | Success principal ->
                LoggedIn principal
```

```
            | Failure err ->
                printfn ".. %A" err
                originalState

    /// do the actions available while you are logged in. Return the new state
    let loggedInActions originalState (principal:IPrincipal) =
        printfn "[%s] Pick a customer to work on. Enter Alice, Bob, or Logout: " princ
ipal.Identity.Name
        let action = Console.ReadLine()

        match action with
        | "Logout"  ->
            // Change state to LoggedOut
            LoggedOut
        // otherwise treat it as a customer name
        | customerName ->
            // Attempt to find customer
            match Authentication.customerIdForName customerName with
            | Success customerId ->
                // found -- change state
                CustomerSelected (principal,customerId)
            | Failure err ->
                // not found -- stay in originalState
                printfn ".. %A" err
                originalState

    let getAvailableCapabilities capabilityProvider customerId principal =
        let getCustomer = capabilityProvider.getCustomer customerId principal
        let updateCustomer = capabilityProvider.updateCustomer customerId principal
        let updatePassword = capabilityProvider.updatePassword customerId principal
        getCustomer,updateCustomer,updatePassword

    /// do the actions available when a selected customer is available. Return the new
 state
    let selectedCustomerActions originalState capabilityProvider customerId principal
=

        // get the individual component capabilities from the provider
        let getCustomerCap,updateCustomerCap,updatePasswordCap =
            getAvailableCapabilities capabilityProvider customerId principal

        // get the text for menu options based on capabilities that are present
        let menuOptionTexts =
            [
            getCustomerCap |> Option.map (fun _ -> "(G)et");
            updateCustomerCap |> Option.map (fun _ -> "(U)pdate");
            updatePasswordCap |> Option.map (fun _ -> "(P)assword");
            ]
            |> List.choose id

        // show the menu
        let actionText =
            match menuOptionTexts with
```

```
            | [] -> " (no other actions available)"
            | texts -> texts |> List.reduce (fun s t -> s + ", " + t)
        printfn "[%s] (D)eselect customer, %s" principal.Identity.Name actionText

        // process the user action
        let action = Console.ReadLine().ToUpper()
        match action with
        | "D" ->
            // revert to logged in with no selected customer
            LoggedIn principal
        | "G" ->
            // use Option.iter in case we don't have the capability
            getCustomerCap
              |> Option.iter BusinessServices.getCustomer
            originalState  // stay in same state
        | "U" ->
            updateCustomerCap
              |> Option.iter BusinessServices.updateCustomer
            originalState
        | "P" ->
            updatePasswordCap
              |> Option.iter BusinessServices.updatePassword
            originalState
        | _ ->
            // unknown option
            originalState

    let rec mainUiLoop capabilityProvider state =
        match state with
        | LoggedOut ->
            let newState = loggedOutActions state
            mainUiLoop capabilityProvider newState
        | LoggedIn principal ->
            let newState = loggedInActions state principal
            mainUiLoop capabilityProvider newState
        | CustomerSelected (principal,customerId) ->
            let newState = selectedCustomerActions state capabilityProvider customerId
 principal
            mainUiLoop capabilityProvider newState
        | Exit ->
            () // done

    let start capabilityProvider  =
        mainUiLoop capabilityProvider LoggedOut
```

Implementing the top-level module

With all this in place, we can implement the top-level module.

This module fetches all the capabilities, adds restrictions as explained previously, and creates a `capabilities` record.

The `capabilities` record is then passed into the user interface when the app is started.

```
module Application=
    open Rop
    open Domain
    open CustomerDatabase
    open Authentication
    open Authorization


    let capabilities =

        let getCustomerOnlyForSameId id principal  =
            onlyForSameId id principal CustomerDatabase.getCustomer

        let getCustomerOnlyForAgentsInBusinessHours id principal =
            let f = CustomerDatabase.getCustomer
            let cap1 = onlyForAgents id principal f
            let restriction f = onlyIfDuringBusinessHours (DateTime.Now) f
            cap1 |> restrict restriction

        let getCustomerOnlyForSameId_OrForAgentsInBusinessHours id principal =
            let cap1 = getCustomerOnlyForSameId id principal
            let cap2 = getCustomerOnlyForAgentsInBusinessHours id principal
            first [cap1; cap2]

        let updateCustomerOnlyForSameId id principal  =
            onlyForSameId id principal CustomerDatabase.updateCustomer

        let updateCustomerOnlyForAgentsInBusinessHours id principal =
            let f = CustomerDatabase.updateCustomer
            let cap1 = onlyForAgents id principal f
            let restriction f = onlyIfDuringBusinessHours (DateTime.Now) f
            cap1 |> restrict restriction

        let updateCustomerOnlyForSameId_OrForAgentsInBusinessHours id principal =
            let cap1 = updateCustomerOnlyForSameId id principal
            let cap2 = updateCustomerOnlyForAgentsInBusinessHours id principal
            first [cap1; cap2]

        let updatePasswordOnlyForSameId id principal =
            let cap = passwordUpdate id principal CustomerDatabase.updatePassword
            cap
            |> Option.map (auditable "UpdatePassword" principal.Identity.Name)

        // create the record that contains the capabilities
        {
        getCustomer = getCustomerOnlyForSameId_OrForAgentsInBusinessHours
        updateCustomer = updateCustomerOnlyForSameId_OrForAgentsInBusinessHours
        updatePassword = updatePasswordOnlyForSameId
        }
```

```
    let start() =
        // pass capabilities to UI
        UserInterface.start capabilities
```

The complete code for this example is available as a gist here.

Summary of Part 2

In part 2, we added authorization and other transforms as a separate concern that could be applied to restrict authority. Again, there is nothing particularly clever about using functions like this, but I hope that this has given you some ideas that might be useful.

Question: Why go to all this trouble? What's the benefit over just testing an "IsAuthorized" flag or something?

Here's a typical use of a authorization flag:

```
if user.CanUpdate() then
   doTheAction()
```

Recall the quote from the previous post: "Capabilities should 'fail safe'. If a capability cannot be obtained, or doesn't work, we must not allow any progress on paths that assumed that it was successful."

The problem with testing a flag like this is that **it's easy to forget, and the compiler won't complain if you do**. And then you have a possible security breach, as in the following code.

```
if user.CanUpdate() then
    // ignore

// now do the action anyway!
doTheAction()
```

Not only that, but by "inlining" the test like this, we're mixing security concerns into our main code, as pointed out earlier.

In contrast, a simple capability approach looks like this:

```
let updateCapability = // attempt to get the capability

match updateCapability with
| Some update -> update()  // call the function
| None -> ()               // can't call the function
```

In this example, it is **not possible to accidentally use the capability** if you are not allowed to, as you literally don't have a function to call! And this has to be handled at compile-time, not at runtime.

Furthermore, as we have just seen, capabilities are just functions, so we get all the benefits of filtering, etc., which are not available with the inlined boolean test version.

Question: In many situations, you don't know whether you can access a resource until you try. So aren't capabilities just extra work?

This is indeed true. For example, you might want to test whether a file exists first, and only then try to access it. The IT gods are always ruthless in these cases, and in the time between checking the file's existence and trying to open it, the file will probably be deleted!

So since we will have to check for exceptions anyway, why do two slow I/O operations when one would have sufficed?

The answer is that the capability model is not about physical or system-level authority, but logical authority -- only having the minimum you need to accomplish a task.

For example, a web service process may be operating at a high level of system authority, and can access any database record. But we don't want to expose that to most of our code. We want to make sure that any failures in programming logic cannot accidentally expose unauthorized data.

Yes, of course, the capability functions themselves must do error handling, and as you can see in the snippets above, I'm using the `Success/Failure` result type as described here. As a result, we will need to merge failures from core functions (e.g. database errors) with capability-specific failures such as `Failure OnlyAllowedOnce`.

Question: You've created a whole module with types defined for each capability. I might have hundreds of capabilities. Do you really expect me to do all this extra work?

There are two points here, so let's address each one in turn:

First, do you have a system that already uses fine-grained authorization, or has business-critical requirements about not leaking data, or performing actions in an unauthorized context, or needs a security audit?

If none of these apply, then indeed, this approach is complete overkill!

But if you *do* have such a system, that raises some new questions:

- should the capabilities that are authorized be explicitly described in the code somewhere?

- and if so, should the capabilities be explicit throughout the code, or only at the top-level (e.g. in the controller) and implicit everywhere else.

The question comes to down to whether you want to be explicit or implicit.

Personally, I prefer things like this to be explicit. It may be a little extra work initially, just a few lines to define each capability, but I find that it generally stops problems from occurring further down the line.

And it has the benefit of acting as a single place to document all the security-related capabilities that you support. Any new requirements would require a new entry here, so can be sure that no capabilities can sneak in under the radar (assuming developers follow these practices).

Question: In this code, you've rolled your own authorization. Wouldn't you use a proper authorization provider instead?

Yes. This code is just an example. The authorization logic is completely separate from the domain logic, so it should be easy to substitute any authorization provider, such as `ClaimsAuthorizationManager` class, or something like XACML.

I've got more questions...

If you missed them, some additional questions are answered at the end of part 1. Otherwise please add your question in the comments below, and I'll try to address it.

Coming up

In the next post, we'll look at how to use types to emulate access tokens and prevent unauthorized access to global functions.

NOTE: All the code for this post is available as a gist here and here.

Using types as access tokens

UPDATE: Slides and video from my talk on this topic

In the previous posts (link, link) we looked at "capabilities" as the basis for locking down code.

But in most of the examples so far, we've been relying on self-discipline to avoid using the global capabilities, or by trying to hide the "raw" capabilities using the `internal` keyword.

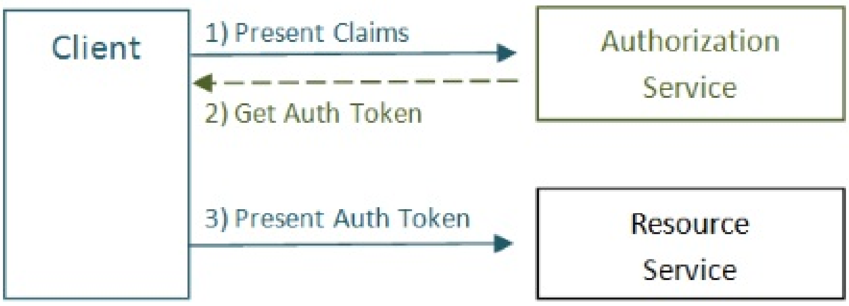
It's a bit ugly -- can we do better?

In this post, we'll show that we can by using types to emulate "access tokens".

Real-world authorization

First, let's step back and look at how authorization works in the real world.

Here's a simplified diagram of a basic authorization system (such as OAuth 2.0).



The steps, in their crudest form, are:

- The client presents some claims to the Authorization Service, including identity and the id and scope (capability) of the service it wants to access.
- The Authorization Service checks whether the client is authorized, and if so, creates an access token which is returned to the client.
- The client then presents this access token to the Resource Service (the service the client wants to use).
- In general, the access token will only let the client do certain things. In our terminology, it has been granted a limited set of capabilities.

Obviously, there's a lot more to it than that, but it will be enough to give us some ideas.

Implementing an Access Token

If we want to emulate this in our design, it's clear that we need some sort of "access token". Since we're running in a single process, and the primary goal is to stop accidental errors, we don't need to do cryptographic signatures and all that. All we need is some object that can *only* be created by an authorization service.

That's easy. We can just use a type with a private constructor!

We'll set it up so that the type can only be created by an Authorization Service, but is required to be passed in to the database service.

For example, here's an F# implementation of the `AccessToken` type. The constructor is private, and there's a static member that returns an instance if authorization is allowed.

```
type AccessToken private() =

    // create an AccessToken that allows access to a particular customer
    static member getAccessToCustomer id principal =
        let principalId = GetIdForPrincipal(principal)
        if (principalId = id) || principal.IsInRole("CustomerAgent") then
            Some <| AccessToken()
        else
            None
```

Next, in the database module, we will add an extra parameter to each function, which is the AccessToken.

Because the AccessToken token is required, we can safely make the database module public now, as no unauthorized client can call the functions.

```
let getCustomer (accessToken:AccessToken) (id:CustomerId) =
    // get customer data

let updateCustomer (accessToken:AccessToken) (id:CustomerId) (data:CustomerData) =
    // update database
```

Note that the accessToken is not actually used in the implementation. It is just there to force callers to obtain a token at compile time.

So let's look at how this might be used in practice.

```
let principal = // from context
let id = // from context

// attempt to get an access token
let accessToken = AuthorizationService.AccessToken.getAccessToCustomer id principal
```

At this point we have an optional access token. Using `Option.map`, we can apply it to `CustomerDatabase.getCustomer` to get an optional capability. And by partially applying the access token, the user of the capability is isolated from the authentication process.

```
let getCustomerCapability =
    accessToken |> Option.map CustomerDatabase.getCustomer
```

And finally, we can attempt to use the capability, if present.

```
match getCustomerCapability with
| Some getCustomer -> getCustomer id
| None -> Failure AuthorizationFailed // error
```

So now we have a statically typed authorization system that will prevent us from accidentally getting too much access to the database.

Oops! We have made a big mistake...

This design looks fine on the surface, but we haven't actually made anything more secure.

The first problem is that the `AccessToken` type is too broad. If I can somehow get hold of an access token for innocently writing to a config file, I might also be able to use it to maliciously update passwords as well.

The second problem is that the `AccessToken` throws away the context of the operation. For example, I might get an access token for updating `CustomerId 1`, but when I actually *use* the capability, I could pass in `CustomerId 2` as the the customer id instead!

The answer to both these issues is to store information in the access token itself, at the point when the authorization is granted.

For example, if the token stores the operation that was requested, the service can check that the token matches the operation being called, which ensures that the token can only be used for that particular operation. In fact, as we'll see in a minute, we can be lazy and have the *compiler* do this checking for us!

And, if we also store any data (such as the customer id) that was part of the authorization request, then we don't need to ask for it again in the service.

What's more, we can trust that the information stored in the token is not forged or tampered with because only the Authorization Service can create the token. In other words, this is the equivalent of the token being "signed".

Revisiting the Access Token design

So let's revisit the design and fix it up.

First we will define a *distinct type* for each capability. The type will also contain any data needed at authorization time, such as the customer id.

For example, here are two types that represent access to capabilities, one for accessing a customer (both read and update), and another one updating a password. Both of these will store the `CustomerId` that was provided at authorization time.

```
type AccessCustomer = AccessCustomer of CustomerId
type UpdatePassword = UpdatePassword of CustomerId
```

Next, the `AccessToken` type is redefined to be a generic container with a `data` field. The constructor is still private, but a public getter is added so clients can access the data field.

```
type AccessToken<'data> = private {data:'data} with
    // but do allow read access to the data
    member this.Data = this.data
```

The authorization implementation is similar to the previous examples, except that this time the capability type and customer id are stored in the token.

```
// create an AccessToken that allows access to a particular customer
let getAccessCustomerToken id principal =
    if customerIdBelongsToPrincipal id principal ||
        principal.IsInRole("CustomerAgent")
    then
        Some {data=AccessCustomer id}
    else
        None

// create an AccessToken that allows access to UpdatePassword
let getUpdatePasswordToken id principal =
    if customerIdBelongsToPrincipal id principal then
        Some {data=UpdatePassword id}
    else
        None
```

Using Access Tokens in the database

With these access token types in place the database functions can be rewritten to require a token of a particular type. The `customerId` is no longer needed as an explicit parameter, because it will be passed in as part of the access token's data.

Note also that both `getCustomer` and `updateCustomer` can use the same type of token (`AccessCustomer`), but `updatePassword` requires a different type (`UpdatePassword`).

```
let getCustomer (accessToken:AccessToken<AccessCustomer>) =
    // get customer id
    let (AccessCustomer id) = accessToken.Data

    // now get customer data using the id
    match db.TryGetValue id with
    | true, value -> Success value
    | false, _ -> Failure (CustomerIdNotFound id)

let updateCustomer (accessToken:AccessToken<AccessCustomer>) (data:CustomerData) =
    // get customer id
    let (AccessCustomer id) = accessToken.Data

    // update database
    db.[id] <- data
    Success ()

let updatePassword (accessToken:AccessToken<UpdatePassword>) (password:Password) =
    Success ()   // dummy implementation
```

Putting it all together

So now let's see all this in action.

The steps to getting a customer are:

- Attempt to get the access token from the authorization service
- If you have the access token, get the `getCustomer` capability from the database
- Finally, if you have the capability, you can use it.

Note that, as always, the `getCustomer` capability does not take a customer id parameter. It was baked in when the capability was created.

```
let principal =  // from context
let customerId = // from context

// attempt to get a capability
let getCustomerCap =
    // attempt to get a token
    let accessToken = AuthorizationService.getAccessCustomerToken customerId principal
    match accessToken with
    // if token is present pass the token to CustomerDatabase.getCustomer,
    // and return a unit->CustomerData
    | Some token ->
        Some (fun () -> CustomerDatabase.getCustomer token)
    | None -> None

// use the capability, if available
match getCustomerCap with
| Some getCustomer -> getCustomer()
| None -> Failure AuthorizationFailed // error
```

Now what happens if we accidentally get the *wrong* type of access token? For example, let us try to access the `updatePassword` function with an `AccessCustomer` token.

```
// attempt to get a capability
let getUpdatePasswordCap =
    let accessToken = AuthorizationService.getAccessCustomerToken customerId principal
    match accessToken with
    | Some token ->
        Some (fun password -> CustomerDatabase.updatePassword token password)
    | None -> None

match getUpdatePasswordCap with
| Some updatePassword ->
    let password = Password "p@ssw0rd"
    updatePassword password
| None ->
    Failure AuthorizationFailed // error
```

This code will not even compile! The line `CustomerDatabase.updatePassword token password` has an error.

```
error FS0001: Type mismatch. Expecting a
    AccessToken<Capabilities.UpdatePassword>
but given a
    AccessToken<Capabilities.AccessCustomer>
The type 'Capabilities.UpdatePassword' does not match the type 'Capabilities.AccessCus
tomer'
```

We have accidentally fetched the wrong kind of Access Token, but we have been stopped from accessing the wrong database method at *compile time*.

Using types in this way is a nice solution to the problem of global access to a potentially dangerous capability.

A complete example in F#

In the last post, I showed a complete console application in F# that used capabilities to update a database.

Now let's update it to use access tokens as well. (The code is available as a gist here).

Since this is an update of the example, I'll focus on just the changes.

Defining the capabilities

The capabilities are as before except that we have defined the two new types (`AccessCustomer` and `UpdatePassword`) to be stored inside the access tokens.

```
module Capabilities =
    open Rop
    open Domain

    // each access token gets its own type
    type AccessCustomer = AccessCustomer of CustomerId
    type UpdatePassword = UpdatePassword of CustomerId

    // capabilities
    type GetCustomerCap = unit -> SuccessFailure<CustomerData,FailureCase>
    type UpdateCustomerCap = CustomerData -> SuccessFailure<unit,FailureCase>
    type UpdatePasswordCap = Password -> SuccessFailure<unit,FailureCase>

    type CapabilityProvider = {
        /// given a customerId and IPrincipal, attempt to get the GetCustomer capabili
ty
        getCustomer : CustomerId -> IPrincipal -> GetCustomerCap option
        /// given a customerId and IPrincipal, attempt to get the UpdateCustomer capab
ility
        updateCustomer : CustomerId -> IPrincipal -> UpdateCustomerCap option
        /// given a customerId and IPrincipal, attempt to get the UpdatePassword capab
ility
        updatePassword : CustomerId -> IPrincipal -> UpdatePasswordCap option
        }
```

Implementing authorization

The authorization implementation must be changed to return `AccessTokens` now. The `onlyIfDuringBusinessHours` restriction applies to capabilities, not access tokens, so it is unchanged.

```
// the constructor is protected
type AccessToken<'data> = private {data:'data} with
    // but do allow read access to the data
    member this.Data = this.data

let onlyForSameId (id:CustomerId) (principal:IPrincipal) =
    if Authentication.customerIdOwnedByPrincipal id principal then
        Some {data=AccessCustomer id}
    else
        None

let onlyForAgents (id:CustomerId) (principal:IPrincipal)  =
    if principal.IsInRole(Authentication.customerAgentRole) then
        Some {data=AccessCustomer id}
    else
        None

let onlyIfDuringBusinessHours (time:DateTime) f =
    if time.Hour >= 8 && time.Hour <= 17 then
        Some f
    else
        None

// constrain who can call a password update function
let passwordUpdate (id:CustomerId) (principal:IPrincipal) =
    if Authentication.customerIdOwnedByPrincipal id principal then
        Some {data=UpdatePassword id}
    else
        None
```

Implementing the database

Compared with the example from the previous post, the database functions have the `CustomerId` parameter replaced with an `AccessToken` instead.

Here's what the database implementation looked like *before* using access tokens:

```
let getCustomer id =
    // code

let updateCustomer id data =
    // code

let updatePassword (id:CustomerId,password:Password) =
    // code
```

And here's what the code looks like *after* using access tokens:

```
let getCustomer (accessToken:AccessToken<AccessCustomer>) =
    // get customer id
    let (AccessCustomer id) = accessToken.Data

    // now get customer data using the id
    // as before

let updateCustomer (accessToken:AccessToken<AccessCustomer>) (data:CustomerData) =
    // get customer id
    let (AccessCustomer id) = accessToken.Data

    // update database
    // as before

let updatePassword (accessToken:AccessToken<UpdatePassword>) (password:Password) =
    // as before
```

Implementing the business services and user interface

The code relating to the business services and UI is completely unchanged.

Because these functions have been passed capabilities only, they are decoupled from both the lower levels and higher levels of the application, so any change in the authorization logic has no effect on these layers.

Implementing the top-level module

The major change in the top-level module is how the capabilities are fetched. We now have an additional step of getting the access token first.

Here's what the code looked like *before* using access tokens:

```
let getCustomerOnlyForSameId id principal  =
    onlyForSameId id principal CustomerDatabase.getCustomer

let getCustomerOnlyForAgentsInBusinessHours id principal =
    let cap1 = onlyForAgents id principal CustomerDatabase.getCustomer
    let restriction f = onlyIfDuringBusinessHours (DateTime.Now) f
    cap1 |> restrict restriction
```

And here's what the code looks like *after* using access tokens:

```
let getCustomerOnlyForSameId id principal  =
    let accessToken = Authorization.onlyForSameId id principal
    accessToken |> tokenToCap CustomerDatabase.getCustomer

let getCustomerOnlyForAgentsInBusinessHours id principal =
    let accessToken = Authorization.onlyForAgents id principal
    let cap1 = accessToken |> tokenToCap CustomerDatabase.getCustomer
    let restriction f = onlyIfDuringBusinessHours (DateTime.Now) f
    cap1 |> restrict restriction
```

The `tokenToCap` function is a little utility that applies the (optional) token to a given function as the first parameter. The output is an (equally optional) capability.

```
let tokenToCap f token =
    token
    |> Option.map (fun token ->
        fun () -> f token)
```

And that's it for the changes needed to support access tokens. You can see all the code for this example here.

Summary of Part 3

In this post, we used types to represent access tokens, as follows:

- The `AccessToken` type is the equivalent of a signed ticket in a distributed authorization system. It has a private constructor and can only be created by the Authorization Service (ignoring reflection, of course!).
- A specific type of `AccessToken` is needed to access a specific operation, which ensures that we can't accidentally do unauthorized activities.
- Each specific type of `AccessToken` can store custom data collected at authorization time, such as a `CustomerId` .
- Global functions, such as the database, are modified so that they cannot be accessed without an access token. This means that they can safely be made public.

Question: Why not also store the caller in the access token, so that no other client can use it?

This is not needed because of the authority-based approach we're using. As discussed in the first post, once a client has a capability, they can pass it around to other people to use, so there is no point limiting it to a specific caller.

Question: The authorization module needs to know about the capability and access token types now. Isn't that adding extra coupling?

If the authorization service is going to do its job, it has to know *something* about what capabilities are available, so there is always some coupling, whether it is implicit ("resources" and "actions" in XACML) or explicit via types, as in this model.

So yes, the authorization service and database service both have a dependency on the set of capabilities, but they are not coupled to each other directly.

Question: How do you use this model in a distributed system?

This model is really only designed to be used in a single codebase, so that type checking can occur.

You could probably hack it so that types are turned into tickets at the boundary, and conversely, but I haven't looked at that at all.

Question: Where can I read more on using types as access tokens?

This type-oriented version of an access token is my own design, although I very much doubt that I'm the first person to think of using types this way. There are some related things for Haskell (example) but I don't know of any directly analogous work that's accessible to mainstream developers.

I've got more questions...

Some additional questions are answered at the end of part 1 and part 2, so read those answers first. Otherwise please add your question in the comments below, and I'll try to address it.

Conclusion

Thanks for making it all the way to the end!

As I said at the beginning, the goal is not to create an absolutely safe system, but instead encourage you to think about and integrate authorization constraints into the design of your system from the beginning, rather than treating it as an afterthought.

What's more, the point of doing this extra work is not just to improve *security*, but also to *improve the general design* of your code. If you follow the principle of least authority, you get modularity, decoupling, explicit dependencies, etc., for free!

In my opinion, a capability-based system works very well for this:

- Functions map well to capabilities, and the need to pass capabilities around fits in very

well with standard functional programming patterns.
- Once created, capabilities hide all the ugliness of authorization from the client, and so the model succeeds in "making security user-friendly by making the security invisible".
- Finally, with the addition of type-checked access tokens, we can have high confidence that no part of our code can access global functions to do unauthorized operations.

I hope you found this series useful, and might inspire you to investigate some of these ideas more fully.

NOTE: All the code for this post is available as a gist here and here.

An introduction to property-based testing

> This post is part of the F# Advent Calendar in English 2014 project. Check out all the other great posts there! And special thanks to Sergey Tihon for organizing this.

UPDATE: I did a talk on property-based testing based on these posts. Slides and video here.

Let's start with a discussion that I hope never to have:

```
Me to co-worker: "We need a function that will add two numbers together, would you min
d implementing it?"
(a short time later)
Co-worker: "I just finished implementing the 'add' function"
Me: "Great, have you written unit tests for it?"
Co-worker: "You want tests as well?" (rolls eyes) "Ok."
(a short time later)
Co-worker: "I just wrote a test. Look! 'Given 1 + 2, I expect output is 3'. "
Co-worker: "So can we call it done now?"
Me: "Well that's only *one* test. How do you know that it doesn't fail for other input
s?"
Co-worker: "Ok, let me do another one."
(a short time later)
Co-worker: "I just wrote a another awesome test. 'Given 2 + 2, I expect output is 4'"
Me: "Yes, but you're still only testing for special cases. How do you know that it doe
sn't fail for other inputs you haven't thought of?"
Co-worker: "You want even *more* tests?"
(mutters "slavedriver" under breath and walks away)
```

But seriously, my imaginary co-worker's complaint has some validity: **How many tests are enough?**

So now imagine that rather than being a developer, you are a test engineer who is responsible for testing that the "add" function is implemented correctly.

Unfortunately for you, the implementation is being written by a burned-out, always lazy and often malicious programmer, who I will call *The Enterprise Developer From Hell*, or "EDFH". (The EDFH has a cousin who you might have heard of).

You are practising test-driven-development, enterprise-style, which means that you write a test, and then the EDFH implements code that passes the test.

So you start with a test like this (using vanilla NUnit style):

```
[<Test>]
let ``When I add 1 + 2, I expect 3``()=
    let result = add 1 2
    Assert.AreEqual(3,result)
```

The EDFH then implements the `add` function like this:

```
let add x y =
    if x=1 && y=2 then
        3
    else
        0
```

And your test passes!

When you complain to the EDFH, they say that they are doing TDD properly, and only writing the minimal code that will make the test pass.

Fair enough. So you write another test:

```
[<Test>]
let ``When I add 2 + 2, I expect 4``()=
    let result = add 2 2
    Assert.AreEqual(4,result)
```

The EDFH then changes the implementation of the `add` function to this:

```
let add x y =
    if x=1 && y=2 then
        3
    else if x=2 && y=2 then
        4
    else
        0
```

When you again complain to the EDFH, they point out that this approach is actually a best practice. Apparently it's called "The Transformation Priority Premise".

At this point, you start thinking that the EDFH is being malicious, and that this back-and-forth could go on forever!

Beating the malicious programmer

So the question is, what kind of test could you write so that a malicious programmer could not create an incorrect implementation, even if they wanted to?

Well, you could start with a much larger list of known results, and mix them up a bit.

```
[<Test>]
let ``When I add two numbers, I expect to get their sum``()=
    for (x,y,expected) in [ (1,2,3); (2,2,4); (3,5,8); (27,15,42); ]
        let actual = add x y
        Assert.AreEqual(expected,actual)
```

But the EDFH is tireless, and will update the implementation to include all of these cases as well.

A much better approach is to generate random numbers and use those for inputs, so that a malicious programmer could not possibly know what to do in advance.

```
let rand = System.Random()
let randInt() = rand.Next()

[<Test>]
let ``When I add two random numbers, I expect their sum``()=
    let x = randInt()
    let y = randInt()
    let expected = x + y
    let actual = add x y
    Assert.AreEqual(expected,actual)
```

If the test looks like this, then the EDFH will be *forced* to implement the `add` function correctly!

One final improvement -- the EDFH might just get lucky and have picked numbers that work by chance, so let's repeat the random number test a number of times, say 100 times.

```
[<Test>]
let ``When I add two random numbers (100 times), I expect their sum``()=
    for _ in [1..100] do
        let x = randInt()
        let y = randInt()
        let expected = x + y
        let actual = add x y
        Assert.AreEqual(expected,actual)
```

So now we're done!

Or are we?

Property based testing

There's just one problem. In order to test the `add` function, you're making use of the `+` function. In other words, you are using one implementation to test another.

In some cases that is acceptable (see the use of "test oracles" in a following post), but in general, it's a bad idea to have your tests duplicate the code that you are testing! It's a waste of time and effort, and now you have two implementations to build and keep up to date.

So if you can't test by using `+`, how *can* you test?

The answer is to create tests that focus on the *properties* of the function -- the "requirements". These properties should be things that are true for *any* correct implementation.

So let's think about what the properties of an `add` function are.

One way of getting started is to think about how `add` differs from other similar functions.

So for example, what is the difference between `add` and `subtract`? Well, for `subtract`, the order of the parameters makes a difference, while for `add` it doesn't.

So there's a good property to start with. It doesn't depend on addition itself, but it does eliminate a whole class of incorrect implementations.

```
[<Test>]
let ``When I add two numbers, the result should not depend on parameter order``()=
    for _ in [1..100] do
        let x = randInt()
        let y = randInt()
        let result1 = add x y
        let result2 = add y x // reversed params
        Assert.AreEqual(result1,result2)
```

That's a good start, but it doesn't stop the EDFH. The EDFH could still implement `add` using `x * y` and this test would pass!

So now what about the difference between `add` and `multiply`? What does addition really mean?

We could start by testing with something like this, which says that `x + x` should the same as `x * 2`:

```
let result1 = add x x
let result2 = x * 2
Assert.AreEqual(result1,result2)
```

But now we are assuming the existence of multiplication! Can we define a property that *only* depends on `add` itself?

One very useful approach is to see what happens when the function is repeated more than once. That is, what if you `add` and then `add` to the result of that?

That leads to the idea that two `add 1`s is the same as one `add 2`. Here's the test:

```
[<Test>]
let ``Adding 1 twice is the same as adding 2``()=
    for _ in [1..100] do
        let x = randInt()
        let y = randInt()
        let result1 = x |> add 1 |> add 1
        let result2 = x |> add 2
        Assert.AreEqual(result1,result2)
```

That's great! `add` works perfectly with this test, while `multiply` doesn't.

However, note that the EDFH could still implement `add` using `y - x` and this test would pass!

Luckily, we have the "parameter order" test above as well. So the combination of both of these tests should narrow it down so that there is only one correct implementation, surely?

After submitting this test suite we find out the EDFH has written an implementation that passes both these tests. Let's have a look:

```
let add x y = 0  // malicious implementation
```

Aarrghh! What happened? Where did our approach go wrong?

Well, we forgot to force the implementation to actually use the random numbers we were generating!

So we need to ensure that the implementation does indeed *do* something with the parameters that are passed into it. We're going to have to check that the result is somehow connected to the input in a specific way.

Is there a trivial property of `add` that we know the answer to without reimplementing our own version?

Yes!

What happens when you add zero to a number? You always get the same number back.

```
[<Test>]
let ``Adding zero is the same as doing nothing``()=
    for _ in [1..100] do
        let x = randInt()
        let result1 = x |> add 0
        let result2 = x
        Assert.AreEqual(result1,result2)
```

So now we have a set of properties that can be used to test any implementation of `add`, and that force the EDFH to create a correct implementation:

Refactoring the common code

There's quite a bit of duplicated code in these three tests. Let's do some refactoring.

First, we'll write a function called `propertyCheck` that does the work of generating 100 pairs of random ints.

`propertyCheck` will also need a parameter for the property itself. This will be a function that takes two ints and returns a bool:

```
let propertyCheck property =
    // property has type: int -> int -> bool
    for _ in [1..100] do
        let x = randInt()
        let y = randInt()
        let result = property x y
        Assert.IsTrue(result)
```

With this in place, we can redefine one of the tests by pulling out the property into a separate function, like this:

```
let commutativeProperty x y =
    let result1 = add x y
    let result2 = add y x // reversed params
    result1 = result2

[<Test>]
let ``When I add two numbers, the result should not depend on parameter order``()=
    propertyCheck commutativeProperty
```

We can also do the same thing for the other two properties.

After the refactoring, the complete code looks like this:

```
let rand = System.Random()
let randInt() = rand.Next()

let add x y = x + y  // correct implementation

let propertyCheck property =
    // property has type: int -> int -> bool
    for _ in [1..100] do
        let x = randInt()
        let y = randInt()
        let result = property x y
        Assert.IsTrue(result)

let commutativeProperty x y =
    let result1 = add x y
    let result2 = add y x // reversed params
    result1 = result2

[<Test>]
let ``When I add two numbers, the result should not depend on parameter order``()=
    propertyCheck commutativeProperty

let adding1TwiceIsAdding2OnceProperty x _ =
    let result1 = x |> add 1 |> add 1
    let result2 = x |> add 2
    result1 = result2

[<Test>]
let ``Adding 1 twice is the same as adding 2``()=
    propertyCheck adding1TwiceIsAdding2OnceProperty

let identityProperty x _ =
    let result1 = x |> add 0
    result1 = x

[<Test>]
let ``Adding zero is the same as doing nothing``()=
    propertyCheck identityProperty
```

Reviewing what we have done so far

We have defined a set of properties that any implementation of `add` should satisfy:

- The parameter order doesn't matter ("commutativity" property)
- Doing `add` twice with 1 is the same as doing `add` once with 2
- Adding zero does nothing ("identity" property)

What's nice about these properties is that they work with *all* inputs, not just special magic numbers. But more importantly, they show us the core essence of addition.

In fact, you can take this approach to the logical conclusion and actually *define* addition as anything that has these properties.

This is exactly what mathematicians do. If you look up addition on Wikipedia, you'll see that it is defined entirely in terms of commutativity, associativity, identity, and so on.

You'll note that in our experiment, we missed defining "associativity", but instead created a weaker property (`x+1+1 = x+2`). We'll see later that the EDFH can indeed write a malicious implementation that satisfies this property, and that associativity is better.

Alas, it's hard to get properties perfect on the first attempt, but even so, by using the three properties we came up with, we have got a much higher confidence that the implementation is correct, and in fact, we have learned something too -- we have understood the requirements in a deeper way.

Specification by properties

A collection of properties like this can be considered a *specification*.

Historically, unit tests, as well as being functional tests, have been used as a sort of specification as well. But an approach to specification using properties instead of tests with "magic" data is an alternative which I think is often shorter and less ambiguous.

You might be thinking that only mathematical kinds of functions can be specified this way, but in future posts, we'll see how this approach can be used to test web services and databases too.

Of course, not every business requirement can be expressed as properties like this, and we must not neglect the social component of software development. Specification by example and domain driven design can play a valuable role when working with non-technical customers.

You also might be thinking that designing all these properties is a lot of work -- and you'd be right! It is the hardest part. In a follow-up post, I'll present some tips for coming up with properties which might reduce the effort somewhat.

But even with the extra effort involved upfront (the technical term for this activity is called "thinking about the problem", by the way) the overall time saved by having automated tests and unambigous specifications will more than pay for the upfront cost later.

In fact, the arguments that are used to promote the benefits of unit testing can equally well be applied to property-based testing! So if a TDD fan tells you that they don't have the time to come up with property-based tests, then they might not be looking at the big picture.

Introducing QuickCheck and FsCheck

We have implemented our own property checking system, but there are quite a few problems with it:

- It only works with integer functions. It would be nice if we could use the same approach for functions that had string parameters, or in fact any type of parameter, including ones we defined ourselves.
- It only works with two parameter functions (and we had to ignore one of them for the `adding1TwiceIsAdding2OnceProperty` and `identity` properties). It would be nice if we could use the same approach for functions with any number of parameters.
- When there is a counter-example to the property, we don't know what it is! Not very helpful when the tests fail!
- There's no logging of the random numbers that we generated, and there's no way to set the seed, which means that we can't debug and reproduce errors easily.
- It's not configurable. For example, we can't easily change the number of loops from 100 to something else.

It would be nice if there was a framework that did all that for us!

Thankfully there is! The "QuickCheck" library was originally developed for Haskell by Koen Claessen and John Hughes, and has been ported to many other languages.

The version of QuickCheck used in F# (and C# too) is the excellent "FsCheck" library created by Kurt Schelfthout. Although based on the Haskell QuickCheck, it has some nice additional features, including integration with test frameworks such as NUnit and xUnit.

So let's look at how FsCheck would do the same thing as our homemade property-testing system.

Using FsCheck to test the addition properties

First, you need to install FsCheck and load the DLL (FsCheck can be a bit finicky -- see the bottom of this page for instructions and troubleshooting).

The top of your script file should look something like this:

```
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)
#I @"Packages\FsCheck.1.0.3\lib\net45"
//#I @"Packages\FsCheck.0.9.2.0\lib\net40-Client"  // use older version for VS2012
#I @"Packages\NUnit.2.6.3\lib"
#r @"FsCheck.dll"
#r @"nunit.framework.dll"

open System
open FsCheck
open NUnit.Framework
```

Once FsCheck is loaded, you can use `Check.Quick` and pass in any "property" function. For now, let's just say that a "property" function is any function (with any parameters) that returns a boolean.

```
let add x y = x + y  // correct implementation

let commutativeProperty (x,y) =
    let result1 = add x y
    let result2 = add y x // reversed params
    result1 = result2

// check the property interactively
Check.Quick commutativeProperty

let adding1TwiceIsAdding2OnceProperty x =
    let result1 = x |> add 1 |> add 1
    let result2 = x |> add 2
    result1 = result2

// check the property interactively
Check.Quick adding1TwiceIsAdding2OnceProperty

let identityProperty x =
    let result1 = x |> add 0
    result1 = x

// check the property interactively
Check.Quick identityProperty
```

If you check one of the properties interactively, say with `Check.Quick commutativeProperty`, you'll see the message:

```
Ok, passed 100 tests.
```

Using FsCheck to find unsatified properties

Let's see what happens when we have a malicious implementation of `add`. In the code below, the EDFH implements `add` as multiplication!

That implementation *will* satisfy the commutative property, but what about the `adding1TwiceIsAdding2OnceProperty`?

```
let add x y =
    x * y // malicious implementation

let adding1TwiceIsAdding2OnceProperty x =
    let result1 = x |> add 1 |> add 1
    let result2 = x |> add 2
    result1 = result2

// check the property interactively
Check.Quick adding1TwiceIsAdding2OnceProperty
```

The result from FsCheck is:

```
Falsifiable, after 1 test (1 shrink) (StdGen (1657127138,295941511)):
1
```

That means that using `1` as the input to `adding1TwiceIsAdding2OnceProperty` will result in `false`, which you can easily see that it does.

The return of the malicious EDFH

By using random testing, we have made it harder for a malicious implementor. They will have to change tactics now!

The EDFH notes that we are still using some magic numbers in the `adding1TwiceIsAdding2OnceProperty` -- namely 1 and 2, and decides to create an implementation that exploits this. They'll use a correct implementation for low input values and an incorrect implementation for high input values:

```
let add x y =
    if (x < 10) || (y < 10) then
        x + y  // correct for low values
    else
        x * y  // incorrect for high values
```

Oh no! If we retest all our properties, they all pass now!

That'll teach us to use magic numbers in our tests!

What's the alternative? Well, let's steal from the mathematicians and create an associative property test.

```
let associativeProperty x y z =
    let result1 = add x (add y z)    // x + (y + z)
    let result2 = add (add x y) z    // (x + y) + z
    result1 = result2

// check the property interactively
Check.Quick associativeProperty
```

Aha! Now we get a falsification:

```
Falsifiable, after 38 tests (4 shrinks) (StdGen (127898154,295941554)):
8
2
10
```

That means that using `(8+2)+10` is not the same as `8+(2+10)`.

Note that not only has FsCheck found some inputs that break the property, but it has found a lowest example. It knows that the inputs `8,2,9` pass but going one higher (`8,2,10`) fails. That's very nice!

Understanding FsCheck: Generators

Now that we have used FsCheck for real, let's pause and have a look at how it works.

The first thing that FsCheck does is generate random inputs for you. This is called "generation", and for each type, there is an associated generator.

For example, to generate a list of sample data, you use the generator along with two parameters: the number of elements in the list and a "size". The precise meaning of "size" depends on the type being generated and the context. Examples of things "size" is used for are: the maximum value of an int; the length of a list; the depth of a tree; etc.

Here's some code that generates ints:

```
// get the generator for ints
let intGenerator = Arb.generate<int>

// generate three ints with a maximum size of 1
Gen.sample 1 3 intGenerator    // e.g. [0; 0; -1]

// generate three ints with a maximum size of 10
Gen.sample 10 3 intGenerator   // e.g. [-4; 8; 5]

// generate three ints with a maximum size of 100
Gen.sample 100 3 intGenerator  // e.g. [-37; 24; -62]
```

In this example, the ints are not generated uniformly, but clustered around zero. You can see this for yourself with a little code:

```
// see how the values are clustered around the center point
intGenerator
|> Gen.sample 10 1000
|> Seq.groupBy id
|> Seq.map (fun (k,v) -> (k,Seq.length v))
|> Seq.sortBy (fun (k,v) -> k)
|> Seq.toList
```

The result is something like this:

```
[(-10, 3); (-9, 14); (-8, 18); (-7, 10); (-6, 27); (-5, 42); (-4, 49);
   (-3, 56); (-2, 76); (-1, 119); (0, 181); (1, 104); (2, 77); (3, 62);
   (4, 47); (5, 44); (6, 26); (7, 16); (8, 14); (9, 12); (10, 3)]
```

You can see that most of the values are in the center (0 is generated 181 times, 1 is generated 104 times), and the outlying values are rare (10 is generated only 3 times).

You can repeat with larger samples too. This one generates 10000 elements in the range [-30,30]

```
intGenerator
|> Gen.sample 30 10000
|> Seq.groupBy id
|> Seq.map (fun (k,v) -> (k,Seq.length v))
|> Seq.sortBy (fun (k,v) -> k)
|> Seq.toList
```

There are plenty of other generator functions available as well as `Gen.sample` (more documentation here).

Understanding FsCheck: Generating all sorts of types automatically

What's great about the generator logic is that it will automatically generate compound values as well.

For example, here is a generator for a tuple of three ints:

```
let tupleGenerator = Arb.generate<int*int*int>

// generate 3 tuples with a maximum size of 1
Gen.sample 1 3 tupleGenerator
// result: [(0, 0, 0); (0, 0, 0); (0, 1, -1)]

// generate 3 tuples with a maximum size of 10
Gen.sample 10 3 tupleGenerator
// result: [(-6, -4, 1); (2, -2, 8); (1, -4, 5)]

// generate 3 tuples with a maximum size of 100
Gen.sample 100 3 tupleGenerator
// result: [(-2, -36, -51); (-5, 33, 29); (13, 22, -16)]
```

Once you have a generator for a base type, `option` and `list` generators follow. Here is a generator for `int option` s:

```
let intOptionGenerator = Arb.generate<int option>
// generate 10 int options with a maximum size of 5
Gen.sample 5 10 intOptionGenerator
// result:  [Some 0; Some -1; Some 2; Some 0; Some 0;
//           Some -4; null; Some 2; Some -2; Some 0]
```

And here is a generator for `int list` s:

```
let intListGenerator = Arb.generate<int list>
// generate 10 int lists with a maximum size of 5
Gen.sample 5 10 intListGenerator
// result:  [ []; []; [-4]; [0; 3; -1; 2]; [1];
//            [1]; []; [0; 1; -2]; []; [-1; -2]]
```

And of course you can generate random strings too!

```
let stringGenerator = Arb.generate<string>

// generate 3 strings with a maximum size of 1
Gen.sample 1 3 stringGenerator
// result: [""; "!"; "I"]

// generate 3 strings with a maximum size of 10
Gen.sample 10 3 stringGenerator
// result: [""; "eiX$a^"; "U%0Ika&r"]
```

The best thing is that the generator will work with your own user-defined types too!

```
type Color = Red | Green of int | Blue of bool

let colorGenerator = Arb.generate<Color>

// generate 10 colors with a maximum size of 50
Gen.sample 50 10 colorGenerator

// result:  [Green -47; Red; Red; Red; Blue true;
//           Green 2; Blue false; Red; Blue true; Green -12]
```

Here's one that generates a user-defined record type containing another user-defined type.

```
type Point = {x:int; y:int; color: Color}

let pointGenerator = Arb.generate<Point>

// generate 10 points with a maximum size of 50
Gen.sample 50 10 pointGenerator

(* result
[{x = -8; y = 12; color = Green -4;};
 {x = 28; y = -31; color = Green -6;};
 {x = 11; y = 27; color = Red;};
 {x = -2; y = -13; color = Red;};
 {x = 6; y = 12; color = Red;};
 // etc
*)
```

There are ways to have more fine-grained control over how your types are generated, but that will have to wait for another post!

Understanding FsCheck: Shrinking

Creating minimum counter-examples is one of the cool things about QuickCheck-style testing.

How does it do this?

There are two parts to the process that FsCheck uses:

First it generates a sequence of random inputs, starting small and getting bigger. This is the "generator" phase as described above.

If any inputs cause the property to fail, it starts "shrinking" the first parameter to find a smaller number. The exact process for shrinking varies depending on the type (and you can override it too), but let's say that for numbers, they get smaller in a sensible way.

For example, let's say that you have a silly property `isSmallerThan80` :

```
let isSmallerThan80 x = x < 80
```

You have generated random numbers and found that then property fails for `100` , and you want to try a smaller number. `Arb.shrink` will generate a sequence of ints, all of which are smaller than 100. Each one of these is tried with the property in turn until the property fails again.

```
isSmallerThan80 100 // false, so start shrinking

Arb.shrink 100 |> Seq.toList
//  [0; 50; 75; 88; 94; 97; 99]
```

For each element in the list, test the property against it until you find another failure:

```
isSmallerThan80 0 // true
isSmallerThan80 50 // true
isSmallerThan80 75 // true
isSmallerThan80 88 // false, so shrink again
```

The property failed with `88` , so shrink again using that as a starting point:

```
Arb.shrink 88 |> Seq.toList
//  [0; 44; 66; 77; 83; 86; 87]
isSmallerThan80 0 // true
isSmallerThan80 44 // true
isSmallerThan80 66 // true
isSmallerThan80 77 // true
isSmallerThan80 83 // false, so shrink again
```

The property failed with `83` now, so shrink again using that as a starting point:

```
Arb.shrink 83 |> Seq.toList
//  [0; 42; 63; 73; 78; 81; 82]
// smallest failure is 81, so shrink again
```

The property failed with `81`, so shrink again using that as a starting point:

```
Arb.shrink 81 |> Seq.toList
//  [0; 41; 61; 71; 76; 79; 80]
// smallest failure is 80
```

After this point, shrinking on 80 doesn't work -- no smaller value will be found.

In this case then, FsCheck will report that `80` falsifies the property and that 4 shrinks were needed.

Just as with generators, FsCheck will generate shrink sequences for almost any type:

```
Arb.shrink (1,2,3) |> Seq.toList
//  [(0, 2, 3); (1, 0, 3); (1, 1, 3); (1, 2, 0); (1, 2, 2)]

Arb.shrink "abcd" |> Seq.toList
//  ["bcd"; "acd"; "abd"; "abc"; "abca"; "abcb"; "abcc"; "abad"; "abbd"; "aacd"]

Arb.shrink [1;2;3] |> Seq.toList
//  [[2; 3]; [1; 3]; [1; 2]; [1; 2; 0]; [1; 2; 2]; [1; 0; 3]; [1; 1; 3]; [0; 2; 3]]
```

And, as with generators, there are ways to customize how shrinking works if needed.

Configuring FsCheck: Changing the number of tests

I mentioned a silly property `isSmallerThan80` -- let's see how FsCheck does with it.

```
// silly property to test
let isSmallerThan80 x = x < 80

Check.Quick isSmallerThan80
// result: Ok, passed 100 tests.
```

Oh dear! FsCheck didn't find a counter-example!

At this point, we can try a few things. First, we can try increasing the number of tests.

We do this by changing the default ("Quick") configuration. There is a field called `MaxTest` that we can set. The default is 100, so let's increase it to 1000.

Finally, to use a specific config, you'll need to use `Check.One(config,property)` rather than just `Check.Quick(property)` .

```
let config = {
    Config.Quick with
        MaxTest = 1000
    }
Check.One(config,isSmallerThan80 )
// result: Ok, passed 1000 tests.
```

Oops! FsCheck didn't find a counter-example with 1000 tests either! Let's try once more with 10000 tests:

```
let config = {
    Config.Quick with
        MaxTest = 10000
    }
Check.One(config,isSmallerThan80 )
// result: Falsifiable, after 8660 tests (1 shrink) (StdGen (539845487,295941658)):
//          80
```

Ok, so we finally got it to work. But why did it take so many tests?

The answer lies in some other configuration settings: `StartSize` and `EndSize` .

Remember that the generators start with small numbers and gradually increase them. This is controlled by the `StartSize` and `EndSize` settings. By default, `StartSize` is 1 and `EndSize` is 100. So at the end of the test, the "size" parameter to the generator will be 100.

But, as we saw, even if the size is 100, very few numbers are generated at the extremes. In this case it means that numbers greater than 80 are unlikely to be generated.

So let's change the `EndSize` to something larger and see what happens!

```
let config = {
    Config.Quick with
        EndSize = 1000
    }
Check.One(config,isSmallerThan80 )
// result: Falsifiable, after 21 tests (4 shrinks) (StdGen (1033193705,295941658)):
//          80
```

That's more like it! Only 21 tests needed now rather than 8660 tests!

Configuring FsCheck: Verbose mode and logging

I mentioned that one of the benefits of FsCheck over a home-grown solution is the logging and reproducibility, so let's have a look at that.

We'll tweak the malicious implementation to have a boundary of `25`. Let's see how FsCheck detects this boundary via logging.

```
let add x y =
    if (x < 25) || (y < 25) then
        x + y  // correct for low values
    else
        x * y  // incorrect for high values

let associativeProperty x y z =
    let result1 = add x (add y z)    // x + (y + z)
    let result2 = add (add x y) z    // (x + y) + z
    result1 = result2

// check the property interactively
Check.Quick associativeProperty
```

The result is:

```
Falsifiable, after 66 tests (12 shrinks) (StdGen (1706196961,295941556)):
1
24
25
```

Again, FsCheck has found that `25` is the exact boundary point quite quickly. But how did it do it?

First, the simplest way to see what FsCheck is doing is to use "verbose" mode. That is, use `Check.Verbose` rather than `Check.Quick`:

```
// check the property interactively
Check.Quick associativeProperty

// with tracing/logging
Check.Verbose associativeProperty
```

When do this, you'll see an output like that shown below. I've added all the comments to explain the various elements.

```
0:    // test 1
-1    // param 1
-1    // param 2
0     // param 3
      // associativeProperty -1 -1 0  => true, keep going
1:    // test 2
0
0
0     // associativeProperty 0 0 0  => true, keep going
2:    // test 3
-2
0
-3    // associativeProperty -2 0 -3  => true, keep going
3:    // test 4
1
2
0     // associativeProperty 1 2 0  => true, keep going
// etc
49:   // test 50
46
-4
50    // associativeProperty 46 -4 50  => false, start shrinking
// etc
shrink:
35
-4
50    // associativeProperty 35 -4 50  => false, keep shrinking
shrink:
27
-4
50    // associativeProperty 27 -4 50  => false, keep shrinking
// etc
shrink:
25
1
29    // associativeProperty 25 1 29  => false, keep shrinking
shrink:
25
1
26    // associativeProperty 25 1 26  => false, keep shrinking
// next shrink fails
Falsifiable, after 50 tests (10 shrinks) (StdGen (995282583,295941602)):
25
1
26
```

This display takes up a lot of space! Can we make it more compact?

Yes -- you can control how each test and shrink is displayed by writing your own custom functions, and telling FsCheck to use them via its `Config` structure.

These functions are generic, and the list of parameters is represented by a list of unknown length (`obj list`). But since I know I am testing a three parameter property I can hard-code a three-element list parameter and print them all on one line.

The configuration also has a slot called `Replay` which is normally `None` , which means that each run will be different.

If you set `Replay` to `Some seed` , then the test will be replayed exactly the same way. The seed looks like `StdGen (someInt,someInt)` and is printed on each run, so if you want to preserve a run all you need to do is paste that seed into the config.

And again, to use a specific config, you'll need to use `Check.One(config,property)` rather than just `Check.Quick(property)` .

Here's the code with the default tracing functions changed, and the replay seed set explicitly.

```
// create a function for displaying a test
let printTest testNum [x;y;z] =
    sprintf "#%-3i %3O %3O %3O\n" testNum x y z

// create a function for displaying a shrink
let printShrink [x;y;z] =
    sprintf "shrink %3O %3O %3O\n" x y z

// create a new FsCheck configuration
let config = {
    Config.Quick with
        Replay = Random.StdGen (995282583,295941602) |> Some
        Every = printTest
        EveryShrink = printShrink
    }

// check the given property with the new configuration
Check.One(config,associativeProperty)
```

The output is now much more compact, and looks like this:

```
#0    -1  -1   0
#1     0   0   0
#2    -2   0  -3
#3     1   2   0
#4    -4   2  -3
#5     3   0  -3
#6    -1  -1  -1
// etc
#46  -21 -25  29
#47  -10  -7 -13
#48   -4 -19  23
#49   46  -4  50
// start shrinking first parameter
shrink  35  -4  50
shrink  27  -4  50
shrink  26  -4  50
shrink  25  -4  50
// start shrinking second parameter
shrink  25   4  50
shrink  25   2  50
shrink  25   1  50
// start shrinking third parameter
shrink  25   1  38
shrink  25   1  29
shrink  25   1  26
Falsifiable, after 50 tests (10 shrinks) (StdGen (995282583,295941602)):
25
1
26
```

So there you go -- it's quite easy to customize the FsCheck logging if you need to.

Let's look at how the shrinking was done in detail. The last set of inputs (46,-4,50) was false, so shrinking started.

```
// The last set of inputs (46,-4,50) was false, so shrinking started
associativeProperty 46 -4 50  // false, so shrink

// list of possible shrinks starting at 46
Arb.shrink 46 |> Seq.toList
// result [0; 23; 35; 41; 44; 45]
```

We'll loop through the list `[0; 23; 35; 41; 44; 45]` stopping at the first element that causes the property to fail:

```
// find the next test that fails when shrinking the x parameter
let x,y,z = (46,-4,50)
Arb.shrink x
|> Seq.tryPick (fun x -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (35, -4, 50)
```

The first element that caused a failure was `x=35`, as part of the inputs `(35, -4, 50)`.

So now we start at 35 and shrink that:

```
// find the next test that fails when shrinking the x parameter
let x,y,z = (35,-4,50)
Arb.shrink x
|> Seq.tryPick (fun x -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (27, -4, 50)
```

The first element that caused a failure was now `x=27`, as part of the inputs `(27, -4, 50)`.

So now we start at 27 and keep going:

```
// find the next test that fails when shrinking the x parameter
let x,y,z = (27,-4,50)
Arb.shrink x
|> Seq.tryPick (fun x -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (26, -4, 50)

// find the next test that fails when shrinking the x parameter
let x,y,z = (26,-4,50)
Arb.shrink x
|> Seq.tryPick (fun x -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (25, -4, 50)

// find the next test that fails when shrinking the x parameter
let x,y,z = (25,-4,50)
Arb.shrink x
|> Seq.tryPick (fun x -> if associativeProperty x y z then None else Some (x,y,z) )
// answer None
```

At this point, `x=25` is as low as you can go. None of its shrink sequence caused a failure. So we're finished with the `x` parameter!

Now we just repeat this process with the `y` parameter

```
// find the next test that fails when shrinking the y parameter
let x,y,z = (25,-4,50)
Arb.shrink y
|> Seq.tryPick (fun y -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (25, 4, 50)

// find the next test that fails when shrinking the y parameter
let x,y,z = (25,4,50)
Arb.shrink y
|> Seq.tryPick (fun y -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (25, 2, 50)

// find the next test that fails when shrinking the y parameter
let x,y,z = (25,2,50)
Arb.shrink y
|> Seq.tryPick (fun y -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (25, 1, 50)

// find the next test that fails when shrinking the y parameter
let x,y,z = (25,1,50)
Arb.shrink y
|> Seq.tryPick (fun y -> if associativeProperty x y z then None else Some (x,y,z) )
// answer None
```

At this point, `y=1` is as low as you can go. None of its shrink sequence caused a failure. So we're finished with the `y` parameter!

Finally, we repeat this process with the `z` parameter

```
// find the next test that fails when shrinking the z parameter
let x,y,z = (25,1,50)
Arb.shrink z
|> Seq.tryPick (fun z -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (25, 1, 38)

// find the next test that fails when shrinking the z parameter
let x,y,z = (25,1,38)
Arb.shrink z
|> Seq.tryPick (fun z -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (25, 1, 29)

// find the next test that fails when shrinking the z parameter
let x,y,z = (25,1,29)
Arb.shrink z
|> Seq.tryPick (fun z -> if associativeProperty x y z then None else Some (x,y,z) )
// answer (25, 1, 26)

// find the next test that fails when shrinking the z parameter
let x,y,z = (25,1,26)
Arb.shrink z
|> Seq.tryPick (fun z -> if associativeProperty x y z then None else Some (x,y,z) )
// answer None
```

And now we're finished with all the parameters!

The final counter-example after shrinking is `(25,1,26)`.

Adding pre-conditions

Let's say that we have a new idea for a property to check. We'll create a property called `addition is not multiplication` which will help to stop any malicious (or even accidental) mixup in the implementations.

Here's our first attempt:

```
let additionIsNotMultiplication x y =
    x + y <> x * y
```

Bt when we run this test, we get a failure!

```
Check.Quick additionIsNotMultiplication
// Falsifiable, after 3 tests (0 shrinks) (StdGen (2037191079,295941699)):
// 0
// 0
```

Well duh, obviously `0+0` and `0*0` are equal. But how can we tell FsCheck to ignore just those inputs and leave all the other ones alone?

This is done via a "condition" or filter expression that is prepended to the property function using `==>` (an operator defined by FsCheck).

Here's an example:

```
let additionIsNotMultiplication x y =
    x + y <> x * y

let preCondition x y =
    (x,y) <> (0,0)

let additionIsNotMultiplication_withPreCondition x y =
    preCondition x y ==> additionIsNotMultiplication x y
```

The new property is `additionIsNotMultiplication_withPreCondition` and can be passed to `Check.Quick` just like any other property.

```
Check.Quick additionIsNotMultiplication_withPreCondition
// Falsifiable, after 38 tests (0 shrinks) (StdGen (1870180794,295941700)):
// 2
// 2
```

Oops! We forgot another case! Let's fix up our precondition again:

```
let preCondition x y =
    (x,y) <> (0,0)
    && (x,y) <> (2,2)

let additionIsNotMultiplication_withPreCondition x y =
    preCondition x y ==> additionIsNotMultiplication x y
```

And now this works.

```
Check.Quick additionIsNotMultiplication_withPreCondition
// Ok, passed 100 tests.
```

This kind of precondition should only be used if you want to filter out a small number of cases.

If most of the inputs will be invalid, then this filtering will be expensive. In this case there is a better way to do it, which will be discussed in a future post.

The FsCheck documentation has more on how you can tweak properties here.

Naming convention for properties

These properties functions have a different purpose from "normal" functions, so how should we name them?

In the Haskell and Erlang world, properties are given a `prop_` prefix by convention. In the .NET world, it is more common to use a suffix like `AbcProperty`.

Also, in F# we have namespaces, modules, and attributes (like `[<Test>]`) that we can use to organize properties and distinguish them from other functions.

Combining multiple properties

Once you have a set of properties, you can combine them into a group (or even, gasp, a *specification*!), by adding them as static members of a class type.

You can then do `Check.QuickAll` and pass in the name of the class.

For example, here are our three addition properties:

```
let add x y = x + y // good implementation

let commutativeProperty x y =
    add x y = add y x

let associativeProperty x y z =
    add x (add y z) = add (add x y) z

let leftIdentityProperty x =
    add x 0 = x

let rightIdentityProperty x =
    add 0 x = x
```

And here's the corresponding static class to be used with `Check.QuickAll`:

```
type AdditionSpecification =
    static member ``Commutative`` x y = commutativeProperty x y
    static member ``Associative`` x y z = associativeProperty x y z
    static member ``Left Identity`` x = leftIdentityProperty x
    static member ``Right Identity`` x = rightIdentityProperty x

Check.QuickAll<AdditionSpecification>()
```

Combining property-based tests with example-based tests

At the beginning of this post, I was dismissive of tests that used "magic" numbers to test a very small part of the input space.

However, I do think that example-based tests have a role that complements property-based tests.

An example-based test is often easier to understand because it is less abstract, and so provides a good entry point and documentation in conjuction with the properties.

Here's an example:

```
type AdditionSpecification =
    static member ``Commutative`` x y = commutativeProperty x y
    static member ``Associative`` x y z = associativeProperty x y z
    static member ``Left Identity`` x = leftIdentityProperty x
    static member ``Right Identity`` x = rightIdentityProperty x

    // some examples as well
    static member ``1 + 2 = 3``() =
        add 1 2 = 3

    static member ``1 + 2 = 2 + 1``() =
        add 1 2 = add 2 1

    static member ``42 + 0 = 0 + 42``() =
        add 42 0 = add 0 42
```

Using FsCheck from NUnit

You can use FsCheck from NUnit and other test frameworks, with an extra plugin (e.g. `FsCheck.NUnit` for Nunit).

Rather than marking a test with `Test` or `Fact`, you use the `Property` attribute. And unlike normal tests, these tests can have parameters!

Here's an example of some tests.

```
open NUnit.Framework
open FsCheck
open FsCheck.NUnit

[<Property(QuietOnSuccess = true)>]
let ``Commutative`` x y =
    commutativeProperty x y

[<Property(Verbose= true)>]
let ``Associative`` x y z =
    associativeProperty x y z

[<Property(EndSize=300)>]
let ``Left Identity`` x =
    leftIdentityProperty x
```

As you can see, you can change the configuration for each test (such as `Verbose` and `EndSize`) via properties of the annotation.

And the `QuietOnSuccess` flag is available to make FsCheck compatible with standard test frameworks, which are silent on success and only show messages if something goes wrong.

Summary

In this post I've introduced you to the basics of property-based checking.

There's much more to cover though! In future posts I will cover topics such as:

- **How to come up with properties that apply to your code**. The properties don't have to be mathematical. We'll look at more general properties such as inverses (for testing serialization/deserialization), idempotence (for safe handling of multiple updates or duplicate messages), and also look at test oracles.
- **How to create your own generators and shrinkers**. We've seen that FsCheck can generate random values nicely. But what about values with constraints such as positive numbers, or valid email addresses, or phone numbers. FsCheck gives you the tools to build your own.
- **How to do model-based testing**, and in particular, how to test for concurrency issues.

I've also introduced the notion of an evil malicious programmer. You might think that such a malicious programmer is unrealistic and over-the-top.

But in many cases *you* act like an unintentionally malicious programmer. You happily create a implementation that works for some special cases, but doesn't work more generally, not out of evil intent, but out of unawareness and blindness.

Like fish unaware of water, we are often unaware of the assumptions we make. Property-based testing can force us to become aware of them.

Until next time -- happy testing!

The code samples used in this post are available on GitHub.

Want more? I have written a follow up post on choosing properties for property-based testing

UPDATE: I did a talk on property-based testing based on these posts. Slides and video here.

Appendix: Installing and troubleshooting FsCheck

The easiest way to make FsCheck available to you is to create an F# project and add the NuGet package "FsCheck.NUnit". This will install both FsCheck and NUnit in the `packages` directory.

If you are using a FSX script file for interactive development, you'll need to load the DLLs from the appropriate package location, like this:

```
// sets the current directory to be same as the script directory
System.IO.Directory.SetCurrentDirectory (__SOURCE_DIRECTORY__)

// assumes nuget install FsCheck.Nunit has been run
// so that assemblies are available under the current directory
#I @"Packages\FsCheck.1.0.3\lib\net45"
//#I @"Packages\FsCheck.0.9.2.0\lib\net40-Client"  // use older version for VS2012
#I @"Packages\NUnit.2.6.3\lib"

#r @"FsCheck.dll"
#r @"nunit.framework.dll"

open System
open FsCheck
open NUnit.Framework
```

Next, test that FsCheck is working correctly by running the following:

```
let revRevIsOrig (xs:list<int>) = List.rev(List.rev xs) = xs

Check.Quick revRevIsOrig
```

If you get no errors, then everything is good.

If you *do* get errors, it's probably because you are on an older version of Visual Studio. Upgrade to VS2013 or failing that, do the following:

- First make sure you have the latest F# core installed (currently 3.1).
- Make sure your that your `app.config` has the appropriate binding redirects.
- Make sure that your NUnit assemblies are being referenced locally rather than from the GAC.

These steps should ensure that compiled code works.

With F# interactive, it can be trickier. If you are not using VS2013, you might run into errors such as `System.InvalidCastException: Unable to cast object of type 'Arrow'`.

The best cure for this is to upgrade to VS2013! Failing that, you can use an older version of FsCheck, such as 0.9.2 (which I have tested successfully with VS2012)

Choosing properties for property-based testing

UPDATE: I did a talk on property-based testing based on these posts. Slides and video here.

In the previous post, I described the basics of property-based testing, and showed how it could save a lot of time by generating random tests.

But here's a common problem. Everyone who sees a property-based testing tool like FsCheck or QuickCheck thinks that it is amazing... but when it times come to start creating your own properties, the universal complaint is: "what properties should I use? I can't think of any!"

The goal of this post is to show some common patterns that can help you discover the properties that are applicable to your code.

Categories for properties

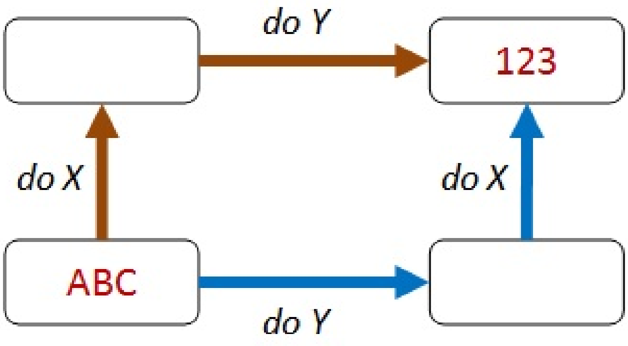
In my experience, many properties can be discovered by using one of the seven approaches listed below.

- "Different paths, same destination"
- "There and back again"
- "Some things never change"
- "The more things change, the more they stay the same"
- "Solve a smaller problem first"
- "Hard to prove, easy to verify"
- "The test oracle"

This is by no means a comprehensive list, just the ones that have been most useful to me. For a different perspective, check out the list of patterns that the PEX team at Microsoft have compiled.

"Different paths, same destination"

These kinds of properties are based on combining operations in different orders, but getting the same result. For example, in the diagram below, doing `X` then `Y` gives the same result as doing `Y` followed by `X`.



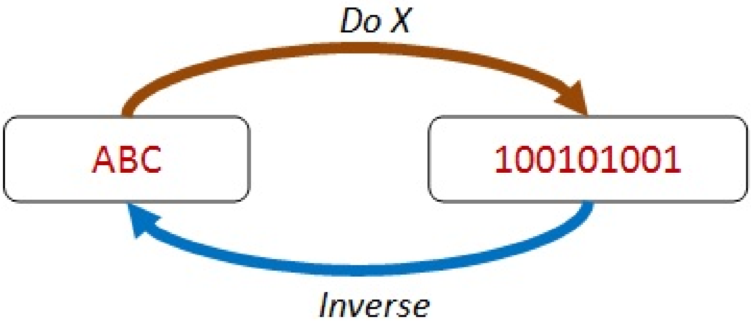
The commutative property of addition is an obvious example of this pattern. For example, the result of `add 1` then `add 2` is the same as the result of `add 2` followed by `add 1`.

This pattern, generalized, can produce a wide range of useful properties. We'll see some more uses of this pattern later in this post.

"There and back again"

These kinds of properties are based on combining an operation with its inverse, ending up with the same value you started with.

In the diagram below, doing `X` serializes `ABC` to some kind of binary format, and the inverse of `X` is some sort of deserialization that returns the same `ABC` value again.



In addition to serialization/deserialization, other pairs of operations can be checked this way: `addition` / `subtraction`, `write` / `read`, `setProperty` / `getProperty`, and so on.

Other pair of functions fit this pattern too, even though they are not strict inverses, pairs such as `insert` / `contains`, `create` / `exists`, etc.

"Some things never change"

These kinds of properties are based on an invariant that is preserved after some transformation.

In the diagram below, the transform changes the order of the items, but the same four items are still present afterwards.

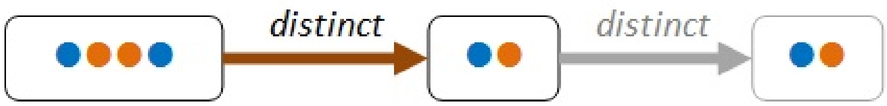


Common invariants include size of a collection (for `map` say), the contents of a collection (for `sort` say), the height or depth of something in proportion to size (e.g. balanced trees).

"The more things change, the more they stay the same"

These kinds of properties are based on "idempotence" -- that is, doing an operation twice is the same as doing it once.

In the diagram below, using `distinct` to filter the set returns two items, but doing `distinct` twice returns the same set again.

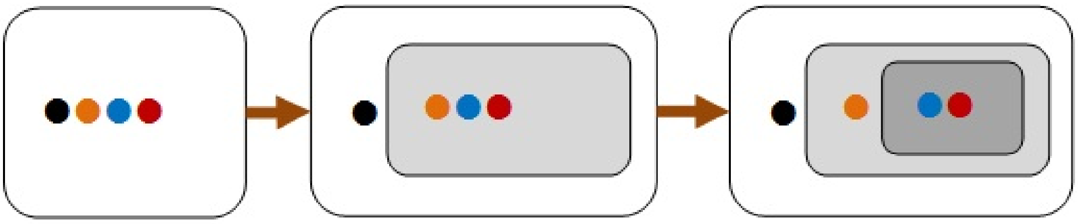


Idempotence properties are very useful, and can be extended to things like database updates and message processing.

"Solve a smaller problem first"

These kinds of properties are based on "structural induction" -- that is, if a large thing can be broken into smaller parts, and some property is true for these smaller parts, then you can often prove that the property is true for a large thing as well.

In the diagram below, we can see that the four-item list can be partitioned into an item plus a three-item list, which in turn can be partitioned into an item plus a two-item list. If we can prove the property holds for two-item list, then we can infer that it holds for the three-item list, and for the four-item list as well.

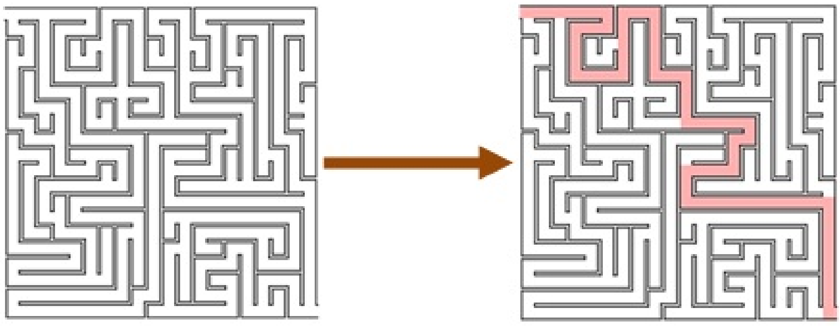


Induction properties are often naturally applicable to recursive structures such as lists and trees.

"Hard to prove, easy to verify"

Often an algorithm to find a result can be complicated, but verifying the answer is easy.

In the diagram below, we can see that finding a route through a maze is hard, but checking that it works is trivial!

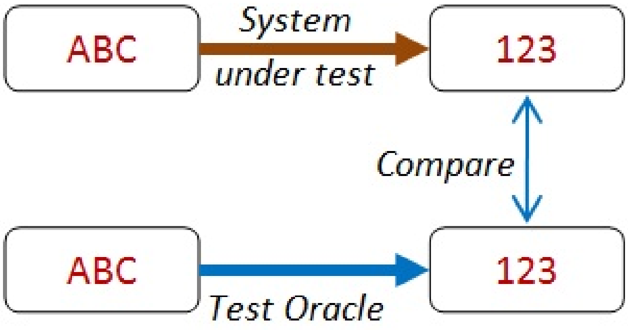


Many famous problems are of this sort, such as prime number factorization. But this approach can be used for even simple problems.

For example, you might check that a string tokenizer works by just concatenating all the tokens again. The resulting string should be the same as what you started with.

"The test oracle"

In many situations you often have an alternate version of an algorithm or process (a "test oracle") that you can use to check your results.



For example, you might have a high-performance algorithm with optimization tweaks that you want to test. In this case, you might compare it with a brute force algorithm that is much slower but is also much easier to write correctly.

Similarly, you might compare the result of a parallel or concurrent algorithm with the result of a linear, single thread version.

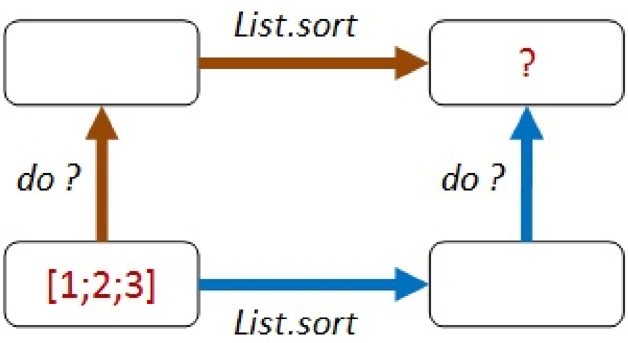
Putting the categories to work with some real examples

In this section, we'll apply these categories to see if we can come up with properties for some simple functions such as "sort a list" and "reverse a list".

"Different paths, same destination" applied to a list sort

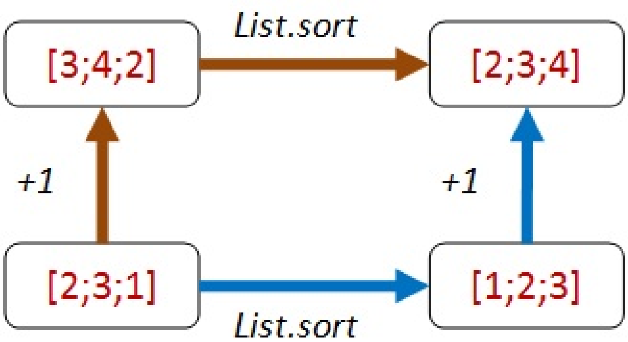
Let's start with *"different paths, same destination"* and apply it to a "list sort" function.

Can we think of any way of combining an operation *before* `List.sort`, and another operation *after* `List.sort`, so that you should end up with the same result? That is, so that "going up then across the top" is the same as "going across the bottom then up".



How about this?

- **Path 1:** We add one to each element of the list, then sort.
- **Path 2:** We sort, then add one to each element of the list.
- Both lists should be equal.



Here's some code that implements that property:

```
let ``+1 then sort should be same as sort then +1`` sortFn aList =
    let add1 x = x + 1

    let result1 = aList |> sortFn |> List.map add1
    let result2 = aList |> List.map add1 |> sortFn
    result1 = result2

// test
let goodSort = List.sort
Check.Quick (``+1 then sort should be same as sort then +1`` goodSort)
// Ok, passed 100 tests.
```

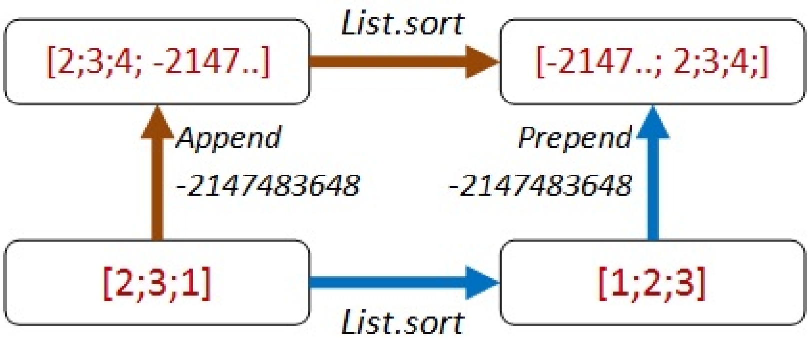
Well, that works, but it also would work for a lot of other transformations too. For example, if we implemented `List.sort` as just the identity, then this property would be satisfied equally well! You can test this for yourself:

```
let badSort aList = aList
Check.Quick (``+1 then sort should be same as sort then +1`` badSort)
// Ok, passed 100 tests.
```

The problem with this property is that it is not exploiting any of the "sortedness". We know that a sort will probably reorder a list, and certainly, the smallest element should be first.

How about adding an item that we *know* will come at the front of the list after sorting?

- **Path 1:** We append `Int32.MinValue` to the *end* of the list, then sort.
- **Path 2:** We sort, then prepend `Int32.MinValue` to the *front* of the list.
- Both lists should be equal.



Here's the code:

```
let ``append minValue then sort should be same as sort then prepend minValue`` sortFn
aList =
    let minValue = Int32.MinValue

    let appendThenSort = (aList @ [minValue]) |> sortFn
    let sortThenPrepend = minValue :: (aList |> sortFn)
    appendThenSort = sortThenPrepend

// test
Check.Quick (``append minValue then sort should be same as sort then prepend minValue`
` goodSort)
// Ok, passed 100 tests.
```

The bad implementation fails now!

```
Check.Quick (``append minValue then sort should be same as sort then prepend minValue`
` badSort)
// Falsifiable, after 1 test (2 shrinks)
// [0]
```

In other words, the bad sort of `[0; minValue]` is *not* the same as `[minValue; 0]`.

So that's good!

But... we've got some hard coded things in there that the Enterprise Developer From Hell (see previous post) could take advantage of! The EDFH will exploit the fact that we always use `Int32.MinValue` and that we always prepend or append it to the test list.

In other words, the EDFH can identify which path we are on and have special cases for each one:

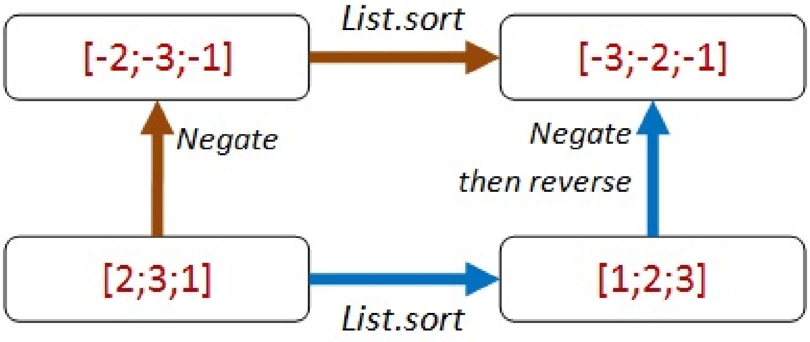
```
// The Enterprise Developer From Hell strikes again
let badSort2 aList =
    match aList with
    | [] -> []
    | _ ->
        let last::reversedTail = List.rev aList
        if (last = Int32.MinValue) then
            // if min is last, move to front
            let unreversedTail = List.rev reversedTail
            last :: unreversedTail
        else
            aList // leave alone
```

And when we check it...

```
// Oh dear, the bad implementation passes!
Check.Quick (``append minValue then sort should be same as sort then prepend minValue`
` badSort2)
// Ok, passed 100 tests.
```

We could fix this by (a) picking a random number smaller than any number in the list and (b) inserting it at a random location rather than always appending it. But rather than getting too complicated, let's stop and reconsider.

An alternative approach which also exploits the "sortedness" is to first negate all the values, then on the path that negates *after* the sort, add an extra reverse as well.




```
let ``negate then sort should be same as sort then negate then reverse`` sortFn aList
=
    let negate x = x * -1

    let negateThenSort = aList |> List.map negate |> sortFn
    let sortThenNegateAndReverse = aList |> sortFn |> List.map negate |> List.rev
    negateThenSort = sortThenNegateAndReverse
```

This property is harder for the EDFH to beat because there are no magic numbers to help identify which path you are on:

```
// test
Check.Quick ( ``negate then sort should be same as sort then negate then reverse`` goo
dSort)
// Ok, passed 100 tests.

// test
Check.Quick ( ``negate then sort should be same as sort then negate then reverse``  ba
dSort)
// Falsifiable, after 1 test (1 shrinks)
// [1; 0]

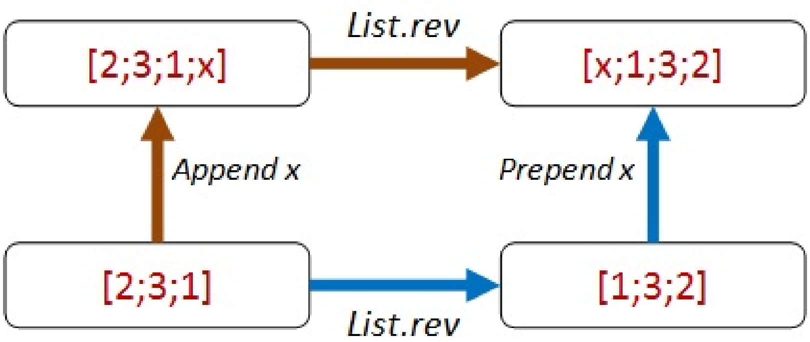
// test
Check.Quick ( ``negate then sort should be same as sort then negate then reverse``  ba
dSort2)
// Falsifiable, after 5 tests (3 shrinks)
// [1; 0]
```

You might argue that we are only testing sorting for lists of integers. But the `List.sort` function is generic and knows nothing about integers per se, so I have high confidence that this property does test the core sorting logic.

Applying "different paths, same destination" to a list reversal function

Ok, enough of `List.sort`. What about applying the same ideas to the list reversal function?

We can do the same append/prepend trick:



Here's the code for the property:

```
let ``append any value then reverse should be same as reverse then prepend same value`
` revFn anyValue aList =

    let appendThenReverse = (aList @ [anyValue]) |> revFn
    let reverseThenPrepend = anyValue :: (aList |> revFn)
    appendThenReverse = reverseThenPrepend
```

Here are the test results for the correct function and for two incorrect functions:

```
// test
let goodReverse = List.rev
Check.Quick (``append any value then reverse should be same as reverse then prepend sa
me value`` goodReverse)
// Ok, passed 100 tests.

// bad implementation fails
let badReverse aList = []
Check.Quick (``append any value then reverse should be same as reverse then prepend sa
me value`` badReverse)
// Falsifiable, after 1 test (2 shrinks)
// true, []

// bad implementation fails
let badReverse2 aList = aList
Check.Quick (``append any value then reverse should be same as reverse then prepend sa
me value`` badReverse2)
// Falsifiable, after 1 test (1 shrinks)
// true, [false]
```

You might notice something interesting here. I never specified the type of the list. The property works with *any* list.

In cases like these, FsCheck will generate random lists of bools, strings, ints, etc.

In both failing cases, the `anyValue` is a bool. So FsCheck is using lists of bools to start with.

Here's an exercise for you: Is this property good enough? Is there some way that the EDFH can create an implementation that will pass?

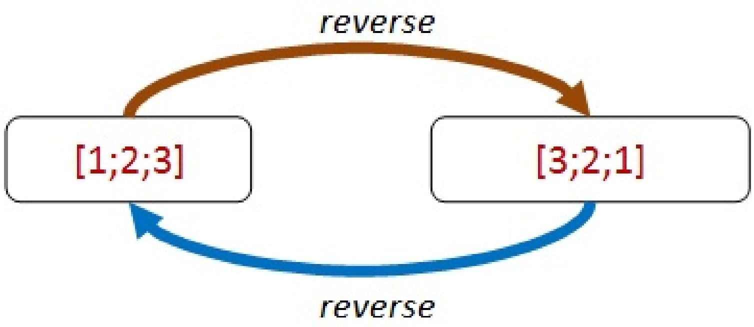
"There and back again"

Sometimes the multi-path style properties are not available or too complicated, so let's look at some other approaches.

We'll start with properties involving inverses.

Let's start with list sorting again. Is there an inverse to sorting? Hmmm, not really. So we'll skip sorting for now.

What about list reversal? Well, as it happens, reversal is its own inverse!



Let's turn that into a property:

```
let ``reverse then reverse should be same as original`` revFn aList =
    let reverseThenReverse = aList |> revFn |> revFn
    reverseThenReverse = aList
```

And it passes:

```
let goodReverse = List.rev
Check.Quick (``reverse then reverse should be same as original`` goodReverse)
// Ok, passed 100 tests.
```

Unfortunately, a bad implementation satisfies the property too!

```
let badReverse aList = aList
Check.Quick (``reverse then reverse should be same as original`` badReverse)
// Ok, passed 100 tests.
```

Nevertheless, the use of properties involving inverses can be very useful to verify that your inverse function (such as deserialization) does indeed "undo" the primary function (such as serialization).

We'll see some real examples of using this in the next post.

"Hard to prove, easy to verify"

So far we've been testing properties without actually caring about the end result of an operation.

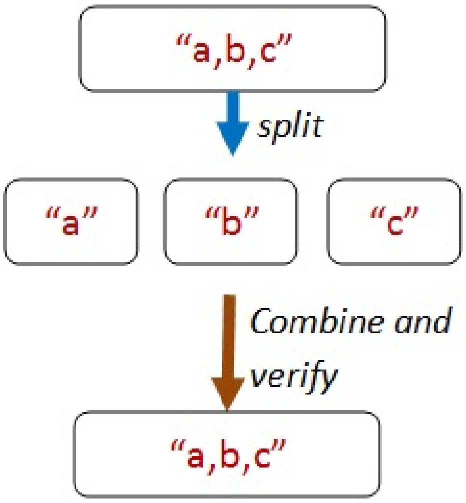
But of course in practice, we do care about the end result!

Now we normally can't really tell if the result is right without duplicating the function under test. But often we can tell that the result is *wrong* quite easily. In the maze diagram from above, we can easily check whether the path works or not.

If we are looking for the *shortest* path, we might not be able to check it, but at least we know that we have *some* valid path.

This principle can be applied quite generally.

For example, let's say that we want to check whether a `string split` function is working. We don't have to write a tokenizer -- all we have to do is ensure that the tokens, when concatenated, give us back the original string!



Here's the core code from that property:

```
let concatWithComma s t = s + "," + t

let tokens = originalString.Split [| ',' |]
let recombinedString =
    // can use reduce safely because there is always at least one token
    tokens |> Array.reduce concatWithComma

// compare the result with the original
originalString = recombinedString
```

But how can we create an original string? The random strings generated by FsCheck are unlikely to contain many commas!

There are ways that you can control exactly how FsCheck generates random data, which we'll look at later.

For now though, we'll use a trick. The trick is to let FsCheck generate a list of random strings, and then we'll build an `originalString` from them by concatting them together.

So here's the complete code for the property:

```
let ``concatting the elements of a string split by commas recreates the original strin
g`` aListOfStrings =
    // helper to make a string
    let addWithComma s t = s + "," + t
    let originalString = aListOfStrings |> List.fold addWithComma ""

    // now for the property
    let tokens = originalString.Split [| ',' |]
    let recombinedString =
        // can use reduce safely because there is always at least one token
        tokens |> Array.reduce addWithComma

    // compare the result with the original
    originalString = recombinedString
```

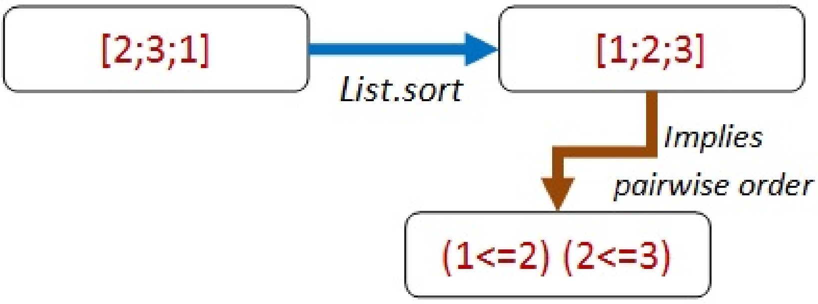
When we test this we are happy:

```
Check.Quick ``concatting the elements of a string split by commas recreates the origin
al string``
// Ok, passed 100 tests.
```

"Hard to prove, easy to verify" for list sorting

So how can we apply this principle to a sorted list? What property is easy to verify?

The first thing that pops into my mind is that for each pair of elements in the list, the first one will be smaller than the second.



So let's make that into a property:

```
let ``adjacent pairs from a list should be ordered`` sortFn aList =
    let pairs = aList |> sortFn |> Seq.pairwise
    pairs |> Seq.forall (fun (x,y) -> x <= y )
```

But something funny happens when we try to check it. We get an error!

```
let goodSort = List.sort
Check.Quick (``adjacent pairs from a list should be ordered`` goodSort)
```

```
System.Exception: Geneflect: type not handled System.IComparable
   at FsCheck.ReflectArbitrary.reflectObj@102-4.Invoke(String message)
   at Microsoft.FSharp.Core.PrintfImpl.go@523-3[b,c,d](String fmt, Int32 len, FSharpFu
nc`2 outputChar, FSharpFunc`2 outa, b os, FSharpFunc`2 finalize, FSharpList`1 args, In
t32 i)
   at Microsoft.FSharp.Core.PrintfImpl.run@521[b,c,d](FSharpFunc`2 initialize, String
fmt, Int32 len, FSharpList`1 args)
```

What does `System.Exception: type not handled System.IComparable` mean? It means that FsCheck is trying to generate a random list, but all it knows is that the elements must be `IComparable`. But `IComparable` is not a type than can be instantiated, so FsCheck throws an error.

How can we prevent this from happening? The solution is to specify a particular type for the property, such as `int list`, like this:

```
let ``adjacent pairs from a list should be ordered`` sortFn (aList:int list) =
    let pairs = aList |> sortFn |> Seq.pairwise
    pairs |> Seq.forall (fun (x,y) -> x <= y )
```

This code works now.

```
let goodSort = List.sort
Check.Quick (``adjacent pairs from a list should be ordered`` goodSort)
// Ok, passed 100 tests.
```

Note that even though the property has been constrained, the property is still a very general one. We could have used `string list` instead, for example, and it would work just the same.

```
let ``adjacent pairs from a string list should be ordered`` sortFn (aList:string list)
 =
    let pairs = aList |> sortFn |> Seq.pairwise
    pairs |> Seq.forall (fun (x,y) -> x <= y )

Check.Quick (``adjacent pairs from a string list should be ordered`` goodSort)
// Ok, passed 100 tests.
```

TIP: If FsCheck throws "type not handled", add explicit type constraints to your property

Are we done now? No! One problem with this property is that it doesn't catch malicious implementations by the EDFH.

```
// bad implementation passes
let badSort aList = []
Check.Quick (``adjacent pairs from a list should be ordered`` badSort)
// Ok, passed 100 tests.
```

Is it a surprise to you that a silly implementation also works?

Hmmm. That tells us that there must be some property *other than pairwise ordering* associated with sorting that we've overlooked. What are we missing here?

This is a good example of how doing property-based testing can lead to insights about design. We thought we knew what sorting meant, but we're being forced to be a bit stricter in our definition.

As it happens, we'll fix this particular problem by using the next principle!

"Some things never change"

A useful kind of property is based on an invariant that is preserved after some transformation, such as preserving length or contents.

They are not normally sufficient in themselves to ensure a correct implementation, but they *do* often act as a counter-check to more general properties.

For example, in the previous post, we created commutative and associative properties for addition, but then noticed that simply having an implementation that returned zero would satisfy them just as well! It was only when we added `x + 0 = x` as a property that we could eliminate that particular malicious implementation.

And in the "list sort" example above, we could satisfy the "pairwise ordered" property with a function that just returned an empty list! How could we fix that?

Our first attempt might be to check the length of the sorted list. If the lengths are different, then the sort function obviously cheated!

```
let ``sort should have same length as original`` sortFn (aList:int list) =
    let sorted = aList |> sortFn
    List.length sorted = List.length aList
```

We check it and it works:

```
let goodSort = List.sort
Check.Quick (``sort should have same length as original`` goodSort )
// Ok, passed 100 tests.
```

And yes, the bad implementation fails:

```
let badSort aList = []
Check.Quick (``sort should have same length as original`` badSort )
// Falsifiable, after 1 test (1 shrink)
// [0]
```

Unfortunately, the BDFH is not defeated and can come up with another compliant implementation! Just repeat the first element N times!

```
// bad implementation has same length
let badSort2 aList =
    match aList with
    | [] -> []
    | head::_ -> List.replicate (List.length aList) head

// for example
// badSort2 [1;2;3]  => [1;1;1]
```

Now when we test this, it passes:

```
Check.Quick (``sort should have same length as original`` badSort2)
// Ok, passed 100 tests.
```

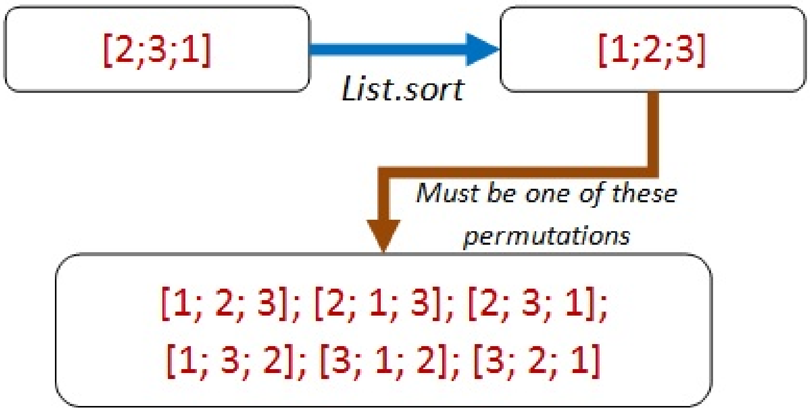
What's more, it also satisfies the pairwise property too!

```
Check.Quick (``adjacent pairs from a list should be ordered`` badSort2)
// Ok, passed 100 tests.
```

Sort invariant - 2nd attempt

So now we have to try again. What is the difference between the real result `[1;2;3]` and the fake result `[1;1;1]` ?

Answer: the fake result is throwing away data. The real result always contains the same contents as the original list, but just in a different order.



That leads us to a new property: a sorted list is always a permutation of the original list. Aha! Let's write the property in terms of permutations now:

```
let ``a sorted list is always a permutation of the original list`` sortFn (aList:int l
ist) =
    let sorted = aList |> sortFn
    let permutationsOfOriginalList = permutations aList

    // the sorted list must be in the seq of permutations
    permutationsOfOriginalList
    |> Seq.exists (fun permutation -> permutation = sorted)
```

Great, now all we need is a permutation function.

Let's head over to StackOverflow and ~~steal~~ borrow an implementation. Here it is:

```
/// given aList and anElement to insert,
/// generate all possible lists with anElement
/// inserted into aList
let rec insertElement anElement aList =
    // From http://stackoverflow.com/a/4610704/1136133
    seq {
        match aList with
        // empty returns a singleton
        | [] -> yield [anElement]
        // not empty?
        | first::rest ->
            // return anElement prepended to the list
            yield anElement::aList
            // also return first prepended to all the sublists
            for sublist in insertElement anElement rest do
                yield first::sublist
        }

/// Given a list, return all permutations of it
let rec permutations aList =
    seq {
        match aList with
        | [] -> yield []
        | first::rest ->
            // for each sub-permutation,
            // return the first inserted into it somewhere
            for sublist in permutations rest do
                yield! insertElement first sublist
        }
```

Some quick interactive tests confirm that it works as expected:

```
permutations ['a';'b';'c'] |> Seq.toList
//  [['a'; 'b'; 'c']; ['b'; 'a'; 'c']; ['b'; 'c'; 'a']; ['a'; 'c'; 'b'];
//  ['c'; 'a'; 'b']; ['c'; 'b'; 'a']]

permutations ['a';'b';'c';'d'] |> Seq.toList
//  [['a'; 'b'; 'c'; 'd']; ['b'; 'a'; 'c'; 'd']; ['b'; 'c'; 'a'; 'd'];
//   ['b'; 'c'; 'd'; 'a']; ['a'; 'c'; 'b'; 'd']; ['c'; 'a'; 'b'; 'd'];
//   ['c'; 'b'; 'a'; 'd']; ['c'; 'b'; 'd'; 'a']; ['a'; 'c'; 'd'; 'b'];
//   ['c'; 'a'; 'd'; 'b']; ['c'; 'd'; 'a'; 'b']; ['c'; 'd'; 'b'; 'a'];
//   ['a'; 'b'; 'd'; 'c']; ['b'; 'a'; 'd'; 'c']; ['b'; 'd'; 'a'; 'c'];
//   ['b'; 'd'; 'c'; 'a']; ['a'; 'd'; 'b'; 'c']; ['d'; 'a'; 'b'; 'c'];
//   ['d'; 'b'; 'a'; 'c']; ['d'; 'b'; 'c'; 'a']; ['a'; 'd'; 'c'; 'b'];
//   ['d'; 'a'; 'c'; 'b']; ['d'; 'c'; 'a'; 'b']; ['d'; 'c'; 'b'; 'a']]

permutations [3;3] |> Seq.toList
//  [[3; 3]; [3; 3]]
```

Excellent! Now let's run FsCheck:

```
Check.Quick (``a sorted list is always a permutation of the original list`` goodSort)
```

Hmmm. That's funny, nothing seems to be happening. And my CPU is maxing out for some reason. What's going on?

What's going on is that you are going to be sitting there for a long time! If you are following along at home, I suggest you right-click and cancel the interactive session now.

The innocent looking `permutations` is really *really* slow for any normal sized list. For example, a list of just 10 items has 3,628,800 permutations. While with 20 items, you are getting to astronomical numbers.

And of course, FsCheck will be doing hundreds of these tests! So this leads to an important tip:

TIP: Make sure your property checks are very fast. You will be running them a LOT!

We've already seen that even in the best case, FsCheck will evaluate the property 100 times. And if shrinking is needed, even more. So make sure your tests are fast to run!

But what happens if you are dealing with real systems such as databases, networks, or other slow dependencies?

In his (highly recommended) video on using QuickCheck, John Hughes tells of when his team was trying to detect flaws in a distributed data store that could be caused by network partitions and node failures.

Of course, killing real nodes thousands of times was too slow, so they extracted the core logic into a virtual model, and tested that instead. As a result, the code was *later refactored* to make this kind of testing easier. In other words, property-based testing influenced the design of the code, just as TDD would.

Sort invariant - 3rd attempt

Ok, so we can't use permutations by just looping through them. So let's use the same idea but write a function that is specific for this case, a `isPermutationOf` function.

```
let ``a sorted list has same contents as the original list`` sortFn (aList:int list) =

    let sorted = aList |> sortFn
    isPermutationOf aList sorted
```

Here's the code for `isPermutationOf` and its associated helper functions:

```
/// Given an element and a list, and other elements previously skipped,
/// return a new list without the specified element.
/// If not found, return None
let rec withoutElementRec anElement aList skipped =
    match aList with
    | [] -> None
    | head::tail when anElement = head ->
        // matched, so create a new list from the skipped and the remaining
        // and return it
        let skipped' = List.rev skipped
        Some (skipped' @ tail)
    | head::tail  ->
        // no match, so prepend head to the skipped and recurse
        let skipped' = head :: skipped
        withoutElementRec anElement tail skipped'

/// Given an element and a list
/// return a new list without the specified element.
/// If not found, return None
let withoutElement x aList =
    withoutElementRec x aList []

/// Given two lists, return true if they have the same contents
/// regardless of order
let rec isPermutationOf list1 list2 =
    match list1 with
    | [] -> List.isEmpty list2 // if both empty, true
    | h1::t1 ->
        match withoutElement h1 list2 with
        | None -> false
        | Some t2 ->
            isPermutationOf t1 t2
```

Let's try the test again. And yes, this time it completes before the heat death of the universe.

```
Check.Quick (``a sorted list has same contents as the original list``  goodSort)
// Ok, passed 100 tests.
```

What's also great is that the malicious implementation now fails to satisfy this property!

```
Check.Quick (``a sorted list has same contents as the original list``  badSort2)
// Falsifiable, after 2 tests (5 shrinks)
// [1; 0]
```

In fact, these two properties, `adjacent pairs from a list should be ordered` and `a sorted list has same contents as the original list` should indeed ensure that any implementation is correct.

Sidebar: Combining properties

Just above, we noted that there were *two* properties needed to define the "is sorted" property. It would be nice if we could combine them into one property `is sorted` so that we can have a single test.

Well, of course we can always merge the two sets of code into one function, but it's preferable to keep functions as small as possible. Furthermore, a property like `has same contents` might be reusable in other contexts as well.

What we want then, is an equivalent to `AND` and `OR` that is designed to work with properties.

FsCheck to the rescue! There are built in operators to combine properties: `.&.` for `AND` and `.|.` for `OR`.

Here is an example of them in use:

```
let ``list is sorted``sortFn (aList:int list) =
    let prop1 = ``adjacent pairs from a list should be ordered`` sortFn aList
    let prop2 = ``a sorted list has same contents as the original list`` sortFn aList
    prop1 .&. prop2
```

When we test the combined property with a good implementation of `sort`, everything works as expected.

```
let goodSort = List.sort
Check.Quick (``list is sorted`` goodSort )
// Ok, passed 100 tests.
```

And if we test a bad implementation, the combined property fails as well.

```
let badSort aList = []
Check.Quick (``list is sorted`` badSort )
// Falsifiable, after 1 test (0 shrinks)
// [0]
```

But there's a problem now. Which of the two properties failed?

What we would like to do is add a "label" to each property so that we can tell them apart. In FsCheck, this is done with the `|@` operator:

```
let ``list is sorted (labelled)``sortFn (aList:int list) =
    let prop1 = ``adjacent pairs from a list should be ordered`` sortFn aList
                |@ "adjacent pairs from a list should be ordered"
    let prop2 = ``a sorted list has same contents as the original list`` sortFn aList
                |@ "a sorted list has same contents as the original list"
    prop1 .&. prop2
```

And now, when we test with the bad sort, we get a message `Label of failing property: a sorted list has same contents as the original list`:

```
Check.Quick (``list is sorted (labelled)`` badSort )
//  Falsifiable, after 1 test (2 shrinks)
//  Label of failing property: a sorted list has same contents as the original list
//  [0]
```

For more on these operators, see the FsCheck documentation under "And, Or and Labels".

And now, back to the property-divising strategies.

"Solving a smaller problem"

Sometimes you have a recursive data structure or a recursive problem. In these cases, you can often find a property that is true of a smaller part.

For example, for a sort, we could say something like:

```
A list is sorted if:
* The first element is smaller (or equal to) the second.
* The rest of the elements after the first element are also sorted.
```

Here is that logic expressed in code:

```
let rec ``First element is <= than second, and tail is also sorted`` sortFn (aList:int
 list) =
    let sortedList = aList |> sortFn
    match sortedList with
    | [] -> true
    | [first] -> true
    | [first;second] ->
        first <= second
    | first::second::tail ->
        first <= second &&
        let subList = second::tail
        ``First element is <= than second, and tail is also sorted`` sortFn subList
```

This property is satisfied by the real sort function:

```
let goodSort = List.sort
Check.Quick (``First element is <= than second, and tail is also sorted`` goodSort )
// Ok, passed 100 tests.
```

But unfortunately, just like previous examples, the malicious implementations also pass.

```
let badSort aList = []
Check.Quick (``First element is <= than second, and tail is also sorted`` badSort )
// Ok, passed 100 tests.

let badSort2 aList =
    match aList with
    | [] -> []
    | head::_ -> List.replicate (List.length aList) head

Check.Quick (``First element is <= than second, and tail is also sorted`` badSort2)
// Ok, passed 100 tests.
```

So as before, we'll need another property (such as the `has same contents` invariant) to ensure that the code is correct.

If you do have a recursive data structure, then try looking for recursive properties. They are pretty obvious and low hanging, when you get the hang of it.

Is the EDFH really a problem?

In the last few examples, I've noted that trivial but wrong implementations often satisfy the properties as well as good implementations.

But should we *really* spend time worrying about this? I mean, if we ever really released a sort algorithm that just duplicated the first element it would be obvious immediately, surely?

So yes, it's true that truly malicious implementations are unlikely to be a problem. On the other hand, you should think of property-based testing not as a *testing* process, but as a *design* process -- a technique that helps you clarify what your system is really trying to do. And if a key aspect of your design is satisfied with just a simple implementation, then perhaps there is something you have overlooked -- something that, when you discover it, will make your design both clearer and more robust.

"The more things change, the more they stay the same"

Our next type of property is "idempotence". Idempotence simply means that doing something twice is the same as doing it once. If I tell you to "sit down" and then tell you to "sit down" again, the second command has no effect.

Idempotence is essential for reliable systems and is a key aspect of service oriented and message-based architectures.

If you are designing these kinds of real-world systems it is well worth ensuring that your requests and processes are idempotent.

I won't go too much into this right now, but let's look at two simple examples.

First, our old friend `sort` is idempotent (ignoring stability) while `reverse` is not, obviously.

```
let ``sorting twice gives the same result as sorting once`` sortFn (aList:int list) =
    let sortedOnce = aList |> sortFn
    let sortedTwice = aList |> sortFn |> sortFn
    sortedOnce = sortedTwice

// test
let goodSort = List.sort
Check.Quick (``sorting twice gives the same result as sorting once`` goodSort )
// Ok, passed 100 tests.
```

In general, any kind of query should be idempotent, or to put it another way: "asking a question should not change the answer".

In the real world, this may not be the case. A simple query on a datastore run at different times may give different results.

Here's a quick demonstration.

First we'll create a `NonIdempotentService` that gives different results on each query.

```
type NonIdempotentService() =
    let mutable data = 0
    member this.Get() =
        data
    member this.Set value =
        data <- value

let ``querying NonIdempotentService after update gives the same result`` value1 value2
 =
    let service = NonIdempotentService()
    service.Set value1

    // first GET
    let get1 = service.Get()

    // another task updates the data store
    service.Set value2

    // second GET called just like first time
    let get2 = service.Get()
    get1 = get2
```

But if we test it now, we find that it does not satisfy the required idempotence property:

```
Check.Quick ``querying NonIdempotentService after update gives the same result``
// Falsifiable, after 2 tests
```

As an alternative, we can create a (crude) `IdempotentService` that requires a timestamp for each transaction. In this design, multiple GETs using the same timestamp will always retrieve the same data.

```
type IdempotentService() =
    let mutable data = Map.empty
    member this.GetAsOf (dt:DateTime) =
        data |> Map.find dt
    member this.SetAsOf (dt:DateTime) value =
        data <- data |> Map.add dt value

let ``querying IdempotentService after update gives the same result`` value1 value2 =
    let service = IdempotentService()
    let dt1 = DateTime.Now.AddMinutes(-1.0)
    service.SetAsOf dt1 value1

    // first GET
    let get1 = service.GetAsOf dt1

    // another task updates the data store
    let dt2 = DateTime.Now
    service.SetAsOf dt2 value2

    // second GET called just like first time
    let get2 = service.GetAsOf dt1
    get1 = get2
```

And this one works:

```
Check.Quick ``querying IdempotentService after update gives the same result``
// Ok, passed 100 tests.
```

So, if you are building a REST GET handler or a database query service, and you want idempotence, you should consider using techniques such as etags, "as-of" times, date ranges, etc.

If you need tips on how to do this, searching for idempotency patterns will turn up some good results.

"Two heads are better than one"

And finally, last but not least, we come to the "test oracle". A test oracle is simply an alternative implementation that gives the right answer, and that you can check your results against.

Often the test oracle implementation is not suitable for production -- it's too slow, or it doesn't parallelize, or it's too poetic, etc., but that doesn't stop it being very useful for testing.

So for "list sort", there are many simple but slow implementations around. For example, here's a quick implementation of insertion sort:

```
module InsertionSort =

    // Insert a new element into a list by looping over the list.
    // As soon as you find a larger element, insert in front of it
    let rec insert newElem list =
        match list with
        | head::tail when newElem > head ->
            head :: insert newElem tail
        | other -> // including empty list
            newElem :: other

    // Sorts a list by inserting the head into the rest of the list
    // after the rest have been sorted
    let rec sort list =
        match list with
        | []   -> []
        | head::tail ->
            insert head (sort tail)

    // test
    // insertionSort  [5;3;2;1;1]
```

With this in place, we can write a property that tests the result against insertion sort.

```
let ``sort should give same result as insertion sort`` sortFn (aList:int list) =
    let sorted1 = aList |> sortFn
    let sorted2 = aList |> InsertionSort.sort
    sorted1 = sorted2
```

When we test the good sort, it works. Good!

```
let goodSort = List.sort
Check.Quick (``sort should give same result as insertion sort`` goodSort)
// Ok, passed 100 tests.
```

And when we test a bad sort, it doesn't. Even better!

```
let badSort aList = aList
Check.Quick (``sort should give same result as insertion sort`` badSort)
// Falsifiable, after 4 tests (6 shrinks)
// [1; 0]
```

Generating Roman numerals in two different ways

We can also use the test oracle approach to cross-check two different implementations when you're not sure that *either* implementation is right!

For example, in my post "Commentary on 'Roman Numerals Kata with Commentary'" I came up with two completely different algorithms for generating Roman Numerals. Can we compare them to each other and test them both in one fell swoop?

The first algorithm was based on understanding that Roman numerals were based on tallying, leading to this simple code:

```
let arabicToRomanUsingTallying arabic =
   (String.replicate arabic "I")
    .Replace("IIIII","V")
    .Replace("VV","X")
    .Replace("XXXXX","L")
    .Replace("LL","C")
    .Replace("CCCCC","D")
    .Replace("DD","M")
    // optional substitutions
    .Replace("IIII","IV")
    .Replace("VIV","IX")
    .Replace("XXXX","XL")
    .Replace("LXL","XC")
    .Replace("CCCC","CD")
    .Replace("DCD","CM")
```

Another way to think about Roman numerals is to imagine an abacus. Each wire has four "unit" beads and one "five" bead.

This leads to the so-called "bi-quinary" approach:

```
let biQuinaryDigits place (unit,five,ten) arabic =
  let digit =  arabic % (10*place) / place
  match digit with
  | 0 -> ""
  | 1 -> unit
  | 2 -> unit + unit
  | 3 -> unit + unit + unit
  | 4 -> unit + five // changed to be one less than five
  | 5 -> five
  | 6 -> five + unit
  | 7 -> five + unit + unit
  | 8 -> five + unit + unit + unit
  | 9 -> unit + ten  // changed to be one less than ten
  | _ -> failwith "Expected 0-9 only"

let arabicToRomanUsingBiQuinary arabic =
  let units = biQuinaryDigits 1 ("I","V","X") arabic
  let tens = biQuinaryDigits 10 ("X","L","C") arabic
  let hundreds = biQuinaryDigits 100 ("C","D","M") arabic
  let thousands = biQuinaryDigits 1000 ("M","?","?") arabic
  thousands + hundreds + tens + units
```

We now have two completely different algorithms, and we can cross-check them with each other to see if they give the same result.

```
let ``biquinary should give same result as tallying`` arabic =
    let tallyResult = arabicToRomanUsingTallying arabic
    let biquinaryResult = arabicToRomanUsingBiQuinary arabic
    tallyResult = biquinaryResult
```

But if we try running this code, we get a `ArgumentException: The input must be non-negative` due to the `String.replicate` call.

```
Check.Quick ``biquinary should give same result as tallying``
// ArgumentException: The input must be non-negative.
```

So we need to only include inputs that are positive. We also need to exclude numbers that are greater than 4000, say, since the algorithms break down there too.

How can we implement this filter?

We saw in the previous post that we could use preconditions. But for this example, we'll try something different and change the generator.

First we'll define a *new* arbitrary integer called `arabicNumber` which is filtered as we want (an "arbitrary" is a combination of a generator algorithm and a shrinker algorithm, as described in the previous post).

```
let arabicNumber = Arb.Default.Int32() |> Arb.filter (fun i -> i > 0 && i <= 4000)
```

Next, we create a new property *which is constrained to only use "arabicNumber"* by using the `Prop.forAll` helper.

We'll give the property the rather clever name of "for all values of arabicNumber, biquinary should give same result as tallying".

```
let ``for all values of arabicNumber biquinary should give same result as tallying`` =

    Prop.forAll arabicNumber ``biquinary should give same result as tallying``
```

Now finally, we can do the cross-check test:

```
Check.Quick ``for all values of arabicNumber biquinary should give same result as tall
ying``
// Ok, passed 100 tests.
```

And we're good! Both algorithms work correctly, it seems.

"Model-based" testing

"Model-based" testing, which we will discuss in more detail in a later post, is a variant on having a test oracle.

The way it works is that, in parallel with your (complex) system under test, you create a simplified model.

Then, when you do something to the system under test, you do the same (but simplified) thing to your model.

At the end, you compare your model's state with the state of the system under test. If they are the same, you're done. If not, either your SUT is buggy or your model is wrong and you have to start over!

Interlude: A game based on finding properties

With that, we have come to the end of the various property categories. We'll go over them one more time in a minute -- but first, an interlude.

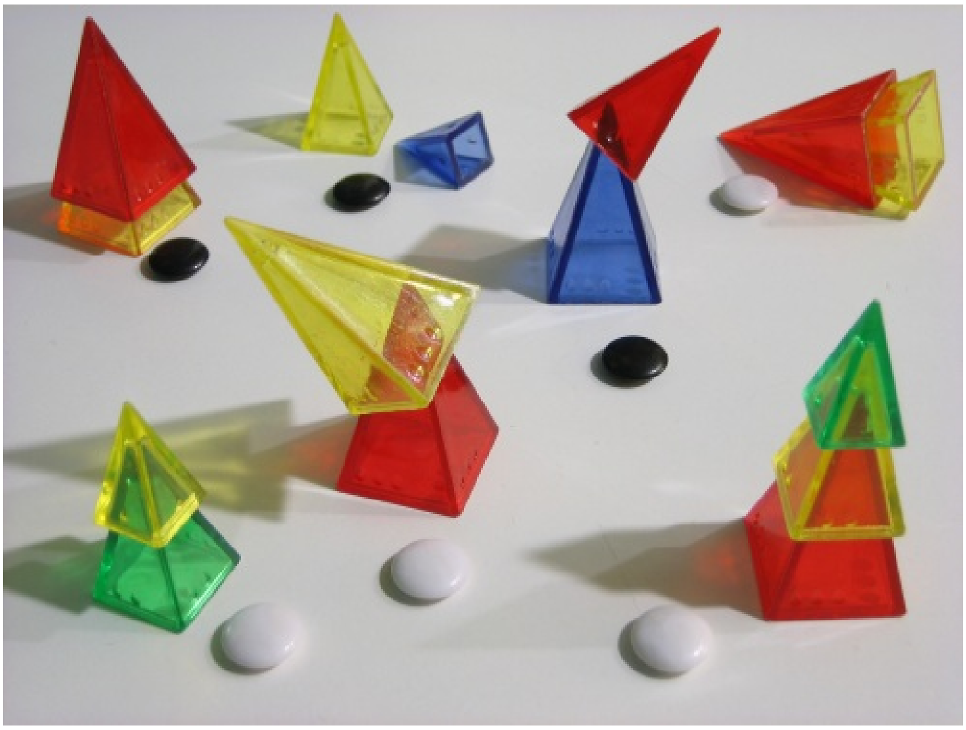
If you sometimes feel that trying to find properties is a mental challenge, you're not alone. Would it help to pretend that it is a game?

As it happens, there *is* a game based on property-based testing.

It's called Zendo and it involves placing sets of objects (such as plastic pyramids) on a table, such that each layout conforms to a pattern -- a rule -- or as we would now say, *a property*!.

The other players then have to guess what the rule (property) is, based on what they can see.

Here's a picture of a Zendo game in progress:



The white stones mean the property has been satisfied, while black stones mean failure. Can you guess the rule here? I'm going to guess that it's something like "a set must have a yellow pyramid that's not touching the ground".

Alright, I suppose Zendo wasn't really inspired by property-based testing, but it is a fun game, and it has even been known to make an appearance at programming conferences.

If you want to learn more about Zendo, the rules are here.

Applying the categories one more time

With all these categories in hand, let's look at one more example problem, and see if we can find properties for it.

This sample is based on the well-known `Dollar` example described in Kent Beck's "TDD By Example" book.

Nat Pryce, of *Growing Object-Oriented Software Guided by Tests* fame, wrote a blog post about property-based testing a while ago ("Exploring Test-Driven Development with QuickCheck").

In it, he expressed some frustration about property-based testing being useful in practice. So let's revisit the example he referenced and see what we can do with it.

We're not going to attempt to critique the design itself and make it more type-driven -- others have done that. Instead, we'll take the design as given and see what properties we can come up with.

So what do we have?

- A `Dollar` class that stores an `Amount` .
- Methods `Add` and `Times` that transform the amount in the obvious way.

```
// OO style class with members
type Dollar(amount:int) =
    member val Amount  = amount with get, set
    member this.Add add =
        this.Amount <- this.Amount + add
    member this.Times multiplier  =
        this.Amount <- this.Amount * multiplier
    static member Create amount  =
        Dollar amount
```

So, first let's try it out interactively to make sure it works as expected:

```
let d = Dollar.Create 2
d.Amount  // 2
d.Times 3
d.Amount  // 6
d.Add 1
d.Amount  // 7
```

But that's just playing around, not real testing. So what kind of properties can we think of?

Let's run through them all again:

- Different paths to same result
- Inverses

- Invariants
- Idempotence
- Structural induction
- Easy to verify
- Test oracle

Let's skip the "different paths" one for now. What about inverses? Are there any inverses we can use?

Yes, the setter and getter form an inverse that we can create a property from:

```
let ``set then get should give same result`` value =
    let obj = Dollar.Create 0
    obj.Amount <- value
    let newValue = obj.Amount
    value = newValue

Check.Quick ``set then get should give same result``
// Ok, passed 100 tests.
```

Idempotence is relevant too. For example, doing two sets in a row should be the same as doing just one. Here's a property for that:

```
let ``set amount is idempotent`` value =
    let obj = Dollar.Create 0
    obj.Amount <- value
    let afterFirstSet = obj.Amount
    obj.Amount <- value
    let afterSecondSet = obj.Amount
    afterFirstSet = afterSecondSet

Check.Quick ``set amount is idempotent``
// Ok, passed 100 tests.
```

Any "structural induction" properties? No, not relevant to this case.

Any "easy to verify" properties? Not anything obvious.

Finally, is there a test oracle? No. Again not relevant, although if we really were designing a complex currency management system, it might be very useful to cross-check our results with a third party system.

Properties for an immutable Dollar

A confession! I cheated a bit in the code above and created a mutable class, which is how most OO objects are designed.

But in "TDD by Example" , Kent quickly realizes the problems with that and changes it to an immutable class, so let me do the same.

Here's the immutable version:

```
type Dollar(amount:int) =
    member val Amount  = amount
    member this.Add add =
        Dollar (amount + add)
    member this.Times multiplier  =
        Dollar (amount * multiplier)
    static member Create amount  =
        Dollar amount

// interactive test
let d1 = Dollar.Create 2
d1.Amount  // 2
let d2 = d1.Times 3
d2.Amount  // 6
let d3 = d2.Add 1
d3.Amount  // 7
```

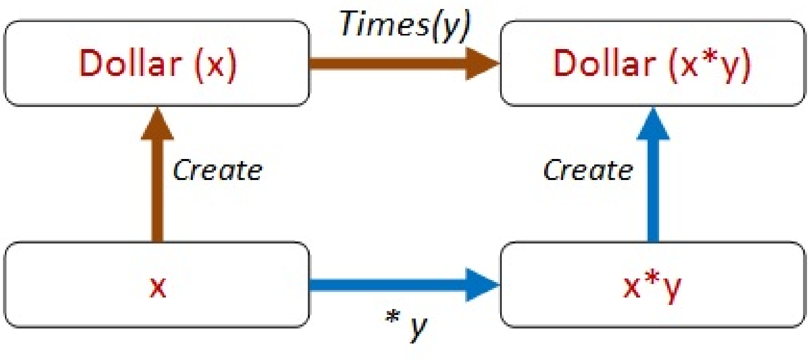
What's nice about immutable code is that we can eliminate the need for testing of setters, so the two properties we just created have now become irrelevant!

To tell the truth they were pretty trivial anyway, so it's no great loss.

So then, what new properties can we devise now?

Let's look at the `Times` method. How can we test that? Which one of the strategies can we use?

I think the "different paths to same result" is very applicable. We can do the same thing we did with "sort" and do a times operation both "inside" and "outside" and see if they give the same result.



Here's that property expressed in code:

```
let ``create then times should be same as times then create`` start multiplier =
    let d0 = Dollar.Create start
    let d1 = d0.Times(multiplier)
    let d2 = Dollar.Create (start * multiplier)
    d1 = d2
```

Great! Let's see if it works!

```
Check.Quick ``create then times should be same as times then create``
// Falsifiable, after 1 test
```

Oops -- it doesn't work!

Why not? Because we forgot that `Dollar` is a reference type and doesn't compare equal by default!

As a result of this mistake, we have discovered a property that we might have overlooked! Let's encode that before we forget.

```
let ``dollars with same amount must be equal`` amount =
    let d1 = Dollar.Create amount
    let d2 = Dollar.Create amount
    d1 = d2

Check.Quick ``dollars with same amount must be equal``
// Falsifiable, after 1 test
```

So now we need to fix this by adding support for `IEquatable` and so on.

You can do that if you like -- I'm going to switch to F# record types and get equality for free!

Dollar properties -- version 3

Here's the `Dollar` rewritten again:

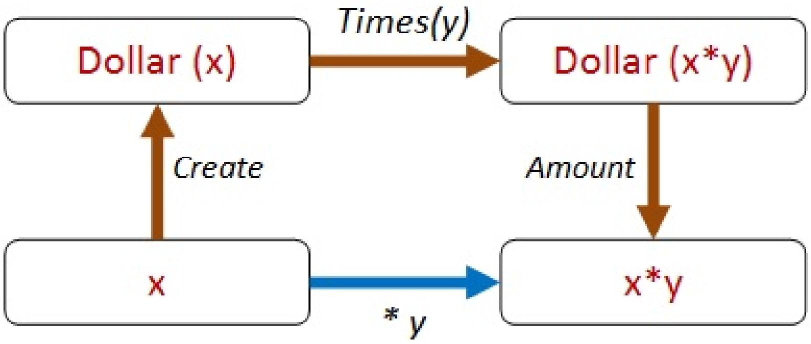
```
type Dollar = {amount:int }
    with
    member this.Add add =
        {amount = this.amount + add }
    member this.Times multiplier  =
        {amount = this.amount * multiplier }
    static member Create amount  =
        {amount=amount}
```

And now our two properties are satisfied:

```
Check.Quick ``dollars with same amount must be equal``
// Ok, passed 100 tests.

Check.Quick ``create then times should be same as times then create``
// Ok, passed 100 tests.
```

We can extend this approach for different paths. For example, we can extract the amount and compare it directly, like this:



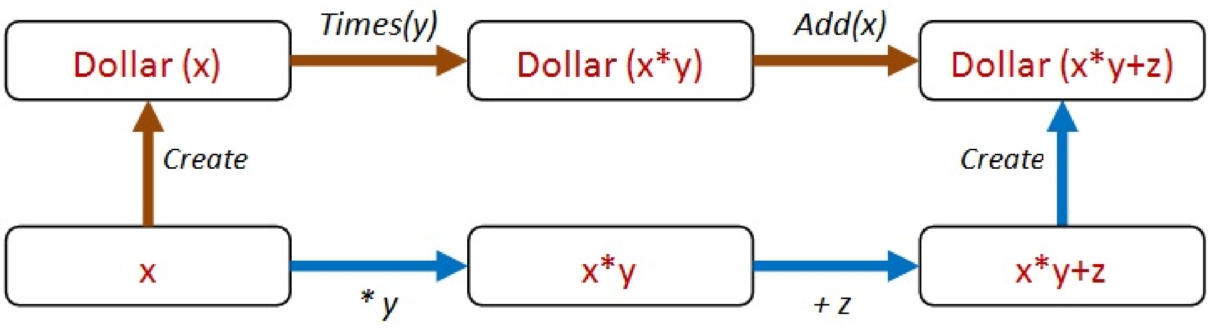
The code looks like this:

```
let ``create then times then get should be same as times`` start multiplier =
    let d0 = Dollar.Create start
    let d1 = d0.Times(multiplier)
    let a1 = d1.amount
    let a2 = start * multiplier
    a1 = a2

Check.Quick ``create then times then get should be same as times``
// Ok, passed 100 tests.
```

And we can also include `Add` in the mix as well.

For example, we can do a `Times` followed by an `Add` via two different paths, like this:



And here's the code:

```
let ``create then times then add should be same as times then add then create`` start
multiplier adder =
    let d0 = Dollar.Create start
    let d1 = d0.Times(multiplier)
    let d2 = d1.Add(adder)
    let directAmount = (start * multiplier) + adder
    let d3 = Dollar.Create directAmount
    d2 = d3

Check.Quick ``create then times then add should be same as times then add then create`
`
// Ok, passed 100 tests.
```

So this "different paths, same result" approach is very fruitful, and we can generate *lots* of paths this way.

Dollar properties -- version 4

Shall we call it done then? I would say not!

We are beginning to get a whiff of a code smell. All this `(start * multiplier) + adder` code seems like a bit of duplicated logic, and could end up being brittle.

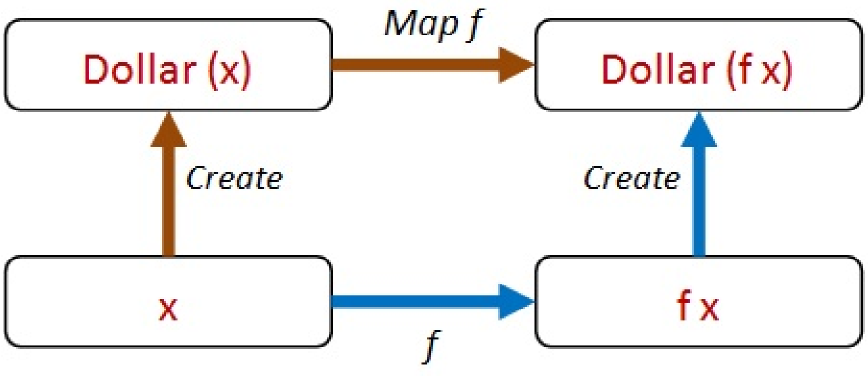
Can we abstract out some commonality that is present all these cases?

If we think about it, our logic is *really* just this:

- Transform the amount on the "inside" in some way.
- Transform the amount on the "outside" in the same way.
- Make sure that the results are the same.

But to test this, the `Dollar` class is going to have to support an arbitrary transform! Let's call it `Map` !

Now all our tests can be reduced to this one property:



Let's add a `Map` method to `Dollar`. And we can also rewrite `Times` and `Add` in terms of `Map`:

```
type Dollar = {amount:int }
    with
    member this.Map f =
        {amount = f this.amount}
    member this.Times multiplier =
        this.Map (fun a -> a * multiplier)
    member this.Add adder =
        this.Map (fun a -> a + adder)
    static member Create amount  =
        {amount=amount}
```

Now the code for our property looks like this:

```
let ``create then map should be same as map then create`` start f =
    let d0 = Dollar.Create start
    let d1 = d0.Map f
    let d2 = Dollar.Create (f start)
    d1 = d2
```

But how can we test it now? What functions should we pass in?

Don't worry! FsCheck has you covered! In cases like this, FsCheck will actually generate random functions for you too!

Try it -- it just works!

```
Check.Quick ``create then map should be same as map then create``
// Ok, passed 100 tests.
```

Our new "map" property is much more general than the original property using "times", so we can eliminate the latter safely.

Logging the function parameter

There's a little problem with the property as it stands. If you want to see what the function is that FsCheck is generating, then Verbose mode is not helpful.

```
Check.Verbose ``create then map should be same as map then create``
```

Gives the output:

```
0:
18
<fun:Invoke@3000>
1:
7
<fun:Invoke@3000>
-- etc
98:
47
<fun:Invoke@3000>
99:
36
<fun:Invoke@3000>
Ok, passed 100 tests.
```

We can't tell what the function values actually were.

However, you can tell FsCheck to show more useful information by wrapping your function in a special `F` case, like this:

```
let ``create then map should be same as map then create2`` start (F (_,f)) =
    let d0 = Dollar.Create start
    let d1 = d0.Map f
    let d2 = Dollar.Create (f start)
    d1 = d2
```

And now when you use Verbose mode...

```
Check.Verbose ``create then map should be same as map then create2``
```

... you get a detailed log of each function that was used:

```
0:
0
{ 0->1 }
1:
0
{ 0->0 }
2:
2
{ 2->-2 }
-- etc
98:
-5
{ -5->-52 }
99:
10
{ 10->28 }
Ok, passed 100 tests.
```

Each `{ 2->-2 }`, `{ 10->28 }`, etc., represents the function that was used for that iteration.

TDD vs. property-based testing

How does property-based testing (PBT) fit in with TDD? This is a common question, so let me quickly give you my take on it.

First off, TDD works with *specific examples*, while PBT works with *universal properties*.

As I said in the previous post, I think examples are useful as a way into a design, and can be a form of documentation. But in my opinion, relying *only* on example-based tests would be a mistake.

Property-based approaches have a number of advantages over example-based tests:

- Property-based tests are more general, and thus are less brittle.
- Property-based tests provide a better and more concise description of requirements than a bunch of examples.
- As a consequence, one property-based test can replace many, many, example-based tests.
- By generating random input, property-based tests often reveal issues that you have overlooked, such as dealing with nulls, missing data, divide by zero, negative numbers, etc.
- Property-based tests force you to think.
- Property-based tests force you to have a clean design.

These last two points are the most important for me. Programming is not a matter of writing lines of code, it is about creating a design that meets the requirements.

So, anything that helps you think deeply about the requirements and what can go wrong should be a key tool in your personal toolbox!

For example, in the Roman Numeral section, we saw that accepting `int` was a bad idea (the code broke!). We had a quick fix, but really we should model the concept of a `PositiveInteger` in our domain, and then change our code to use that type rather than just an `int`. This demonstrates how using PBT can actually improve your domain model, not just find bugs.

Similarly, introducing a `Map` method in the Dollar scenario not only made testing easier, but actually improved the usefulness of the Dollar "api".

Stepping back to look at the big picture, though, TDD and property-based testing are not at all in conflict. They share the same goal of building correct programs, and both are really more about design than coding (think "Test-driven *design*" rather than "Test-driven *development*").

The end, at last

So that brings us to the end of another long post on property-based testing!

I hope that you now have some useful approaches that you can take away and apply to your own code base.

Next time, we'll look at some real-world examples, and how you can create custom generators that match your domain.

The code samples used in this post are available on GitHub.

Commentary on 'Roman Numerals Kata with Commentary'

I recently watched a video called "Roman Numerals Kata with Commentary". In it, Corey Haines demonstrates how to implement the Arabic to Roman Numerals Kata in Ruby using a TDD approach.

This video annoyed me intensely.

I mean no disrespect to Corey Haines' programming skills, and many people seem to have found the video useful, but I just found it exasperating.

I thought that in this post I'd try to explain why I got annoyed, and to present my alternative approach to problems like this.

Where are the requirements?

> *"Few programmers write even a rough sketch of what their programs will do before they start coding. Most programmers regard anything that doesn't generate code to be a waste of time."*
>
> *Leslie Lamport, "Why We Should Build Software Like We Build Houses"*

In standard TDD fashion, the video starts with implementing a initial failing case (handling zero), then making that work, then adding a test case for handling "1", then making that work, and so on.

This was the first thing that irritated me -- diving into code without really understanding the requirements.

A programming kata) is so called because the goal is to practice your skills as a developer. But for me, coding skills are just one aspect of a being a software developer, and not always the most important.

If there is anything that needs practicing by most developers, it is listening to and understanding the needs of the customer (a.k.a. requirements). We should never forget that our goal is to deliver value, not just to write code.

In this case, even though there is a wiki page for the kata, the requirements are still somewhat fuzzy, and so I view this as an excellent opportunity to drill down into them, and maybe learn something new.

Becoming a domain expert

In fact, I believe that going as deep as possible into the requirements has some important benefits.

It's fun. It's fun to really understand a new domain. I like to learn new things -- it is one of the perks of being a developer.

It's not just me. Dan North tells of how much fun he had working very closely with domain experts in his "accelerating agile" presentation. Part of that team's success was that the developers studied the domain (trading) right along with the traders themselves, so that communication was easy and confusion minimized.

Good design. I do believe that in order to produce good software you have to become reasonably expert in the domain you are attempting to model. This is the thesis behind Domain Driven Design, of course, but also is a key component of an Agile process: the "on site customer" who works very closely with the developers at all stages.

And almost always, understanding the requirements properly will lead you into the right* way to implement a solution. No amount of shallow iterations will make up for a lack of deep insight.

* Of course there is not really a "right" way, but there are plenty of wrong ways. So here I just mean not horribly complicated and unmaintainable.

Good tests. You can't create good tests without understanding the requirements. A process like BDD makes this explicit; the requirements are written in such a way that they actually *become* the tests.

Understanding Roman numerals

> *"During an inception, when we are most ignorant about most aspects of the project, the best use we can possibly make of the time available is to attempt to identify and reduce our ignorance across all the axes we can think of."* -- Dan North, *"Deliberate Discovery"*

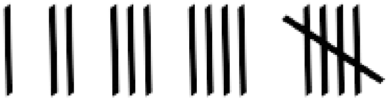
So, how does this apply to the Roman Numerals Kata? Should we seriously become domain experts before we write a line of code?

I would say yes!

I know it is a trivial problem, and it seems like overkill, but then again, this is a kata, so you should be practising all the steps carefully and mindfully.

So, what can we find out about Roman numerals?

First, a little background reading from a reliable source shows that they probably originated from something similar to tally marks.



This explains the simple strokes for "I" to "IIII" and then the different symbol for "V".

As it evolved, symbols were added for ten and fifty, one hundred and five hundred, and so on.
This system of counting with ones and fives can be seen in the design of the abacus, old and new.




In fact, this system even has a name which I'd never heard of -- "bi-quinary coded decimal". Isn't that fun? I shall now attempt to drop that phrase into casual conversation wherever possible. (And by the way, the little stones used as counters are called "calculi", whence the name for the bane of high school students everywhere.)

Much later, in the 13th century, certain abbreviations were added -- substituting "IV" for "IIII" and "IX" for "VIIII". This subtractive notation means that the order of the symbols becomes important, something that is not required for a pure tally-based system.

These new requirements show us that nothing has changed in the development biz...

Pope: "We need to add subtractive notation ASAP -- the Arabs are beating us on features."
You: "But it's not backwards compatible, sir. It's a breaking change!"
Pope: "Tough. I need it by next week."

So now that we know all about Roman numerals, do we have enough information to create the requirements?

Alas, no. As we investigate further, it becomes clear that there is a lot of inconsistency. There is no ISO or ANSI standard for Roman numerals!

This is not unusual of course. A fuzziness around requirements affects most software projects. Indeed, part of our job as a developer is to help clarify things and eliminate ambiguity. So, let's create some requirements based on what we know so far.

The requirements for this kata

> *"Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis." -- Alan Perlis, Epigrams*

I think we would all agree that having unambiguous and testable requirements is a critical step towards having a successful project.

Now when I talk about "requirements", I'm not talking about a 200 page document that takes six months to write. I'm just talking about a few bullet points that take 5 or 10 minutes to write down.

But... it is important to have them. Thinking carefully before coding is an essential skill that needs to be practiced, and so I would recommend doing this step as part of the discipline for any code kata.

So here are the requirements as I see them:

- The output will be generated by tallying 1, 5, 10, 50, 100, 500, and 1000, using the symbols "I", "V", "X", "L", "C", "D" and "M" respectively.
- The symbols must be written in descending order: "M" before "D" before "C" before "L", etc.
- Using the tallying logic, it's clear that we can only have up to four repetitions of "I", "X", "C" and "M". And only one "V", "L" or "D". Any more than that and the multiple tally marks are abbreviated to the next "higher" tally mark.
- Finally, we have the six (optional) substitution rules: "IIII"=>"IV", "VIIII"=>"IX", "XXXX"=>"XL", "LXXXX"=>"XC", "CCCC"=>"CD", "DCCCC"=>"CM". These are exceptions to the descending order rule.

There's one other very important requirement that isn't on this list.

- What is the range of valid inputs?

If we don't explicitly document this, we could easily assume that all integers are valid, including zero and negative numbers.

And what about large numbers, in the millions or billions? Are they allowed? Probably not.

So let's be explicit and say that the valid input ranges from 0 to 4000. Then what should happen if the input is not valid? Return an empty string? Throw an exception?

In a functional programming language like F#, the most common approach is to return an `Option` type, or to return a Success/Failure `Choice` type. Let's just use an `Option`, so to finish off the requirements, we have:

- The Arabic number 0 is mapped to the empty string.
- If the input is < 0 or > 4000 return `None` otherwise return `Some(roman)`, where `roman` is the Arabic number converted to Roman numerals as described above.

So to sum up this step, we have read about Roman numerals, learned a few fun things, and come up with some clear requirements for the next stage. The whole thing took only 5-10 mins. In my opinion, that was time well spent.

Writing the tests

> *"Unit tests have been compared with shining a flashlight into a dark room in search of a monster. Shine the light into the room and then into all the scary corners. It doesn't mean the room is monster free --- just that the monster isn't standing where you've shined your flashlight."*

Now that we have the requirements, we can start writing the tests.

In the original video, the tests were developed incrementally, starting with 0, then 1, and so on.

Personally, I think that there are a number of problems with that approach.

First, as we should know, a major goal of TDD is not testing but *design*.

But this micro, incremental approach to design does not seem to me to lead to a particularly good end result.

For example, in the video, there is a big jump in the implementation complexity from testing the "I" case to testing the "II" case. But the rationale is a bit hard to understand, and to me it smacks a little of sleight-of-hand, of someone who already knows the answer, rather than naturally evolving from the previous case.

Unfortunately, I have seen this happen a lot with a strict TDD approach. You might be cruising along nicely and then bump into a huge road block which forces a huge rethink and refactoring.

A strict TDD'er using Uncle Bob's "Transformation Priority Premise" approach would say that that is fine and good, and part of the process.

Personally, I'd rather start with the trickiest requirements first, and front-load the risk rather than leaving it till the end.

Second, I don't like testing individual cases. I'd prefer that my tests cover *all* inputs. This is not always feasible, but when you can do it, as in this case, I think you should.

Two tests compared

To demonstrate what I mean, let's compare the test suite developed in the video with a more general requirements-based test.

The test suite developed in the video checks only the obvious inputs, plus the case 3497 "for good measure". Here's the Ruby code ported to F#:

```
[<Test>]
let ``For certain inputs, expect certain outputs``() =
    let testpairs = [
      (1,"I")
      (2,"II")
      (4,"IV")
      (5,"V")
      (9,"IX")
      (10,"X")
      // etc
      (900,"CM")
      (1000,"M")
      (3497,"MMMCDXCVII")
      ]
    for (arabic,expectedRoman) in testpairs do
       let roman = arabicToRoman arabic
       Assert.AreEqual(expectedRoman, roman)
```

With this set of inputs, how confident are we that the code meets the requirements?

In a simple case like this, I might be reasonably confident, but this approach to testing worries me because of the use of "magic" test inputs that are undocumented.

For example, why was 3497 plucked out of nowhere? Because (a) it is bigger than a thousand and (b) it has some 4's and 9's in it. But the reason it was picked is not documented in the test code.

Furthermore, if we compare this test suite with the requirements, we can see that the second and third requirements are not explicitly tested for at all. True, the test with 3497 implicitly checks the ordering requirement ("M" before "C" before "X"), but that is never made explicit.

Now compare that test with this one:

```
[<Test>]
let ``For all valid inputs, there must be a max of four "I"s in a row``() =
    for i in [1..4000] do
       let roman = arabicToRoman i
       roman |> assertMaxRepetition "I" 4
```

This test checks the requirement that you can only have four repetitions of "I".

Unlike the one in the TDD video, this test case covers *all possible inputs*, not just one. If it passes, I will have complete confidence that the code meets this particular requirement.

Property-based testing

If you are not familiar with this approach to testing, it is called *"property-based testing"*. You define a "property" that must be true in general, and then you generate as many inputs as possible in order to find cases where the property is not true.

In this case, we can test all 4000 inputs. In general though, our problems have a much larger range of possible inputs, so we generally just test on some representative sample of the inputs.

Most property-based testing tools are modelled after Haskell's QuickCheck, which is a tool that automatically generates "interesting" inputs for you, in order to find edge cases as quickly as possible. These inputs would include things like nulls, negative numbers, empty lists, strings with non-ascii characters in them, and so on.

An equivalent to QuickCheck is available for most languages now, including FsCheck for F#.

The advantage of property-based testing is that it forces you to think about the requirements in general terms, rather than as lots of special cases.

That is, rather than a test that says `the input "4" maps to "IV"`, we have a more general test that says `any input with 4 in the units place has "IV" as the last two characters`.

Implementing a property-based test

To switch to property-based testing for the requirement above, I would refactor the code so that (a) I create a function that defines a property and then (b) I check that property against a range of inputs.

Here's the refactored code:

```
// Define a property that should be true for all inputs
let ``has max rep of four Is`` arabic =
   let roman = arabicToRoman arabic
   roman |> assertMaxRepetition "I" 4

// Explicitly enumerate all inputs...
[<Test>]
let ``For all valid inputs, there must be a max of four "I"s``() =
    for i in [1..4000] do
       //check that the property holds
       ``has max rep of four Is`` i

// ...Or use FsCheck to generate inputs for you
let isInRange i = (i >= 1) && (i <= 4000)
// input is in range implies has max of four Is
let prop i = isInRange i ==> ``has max rep of four Is`` i
// check all inputs for this property
Check.Quick prop
```

Or for example, let's say that I want to test the substitution rule for 40 => "XL".

```
// Define a property that should be true for all inputs
let ``if arabic has 4 tens then roman has one XL otherwise none`` arabic =
   let roman = arabicToRoman arabic
   let has4Tens = (arabic % 100 / 10) = 4
   if has4Tens then
       assertMaxOccurs "XL" 1 roman
   else
       assertMaxOccurs "XL" 0 roman

// Explicitly enumerate all inputs...
[<Test>]
let ``For all valid inputs, check the XL substitution``() =
    for i in [1..4000] do
       ``if arabic has 4 tens then roman has one XL otherwise none`` i

// ...Or again use FsCheck to generate inputs for you
let isInRange i = (i >= 1) && (i <= 4000)
let prop i = isInRange i ==> ``if arabic has 4 tens then roman has one XL otherwise no
ne`` i
Check.Quick prop
```

I'm not going to go into property-based testing any more here, but I think you can see the benefits over hand-crafted cases with magic inputs.

The code for this post has a full property-based test suite.

Requirements Driven Design?

At this point, we can start on the implementation.

Unlike the TDD video, I'd rather build the implementation by iterating on the *requirements*, not on the *test cases*. I need a catchy phrase for this, so I'll call it Requirements Driven Design?. Watch out for a Requirements Driven Design Manifesto coming soon.

And rather than implementing code that handles individual inputs one by one, I prefer my implementations to cover as many input cases as possible -- preferably all of them. As each new requirement is added the implementation is modified or refined, using the tests to ensure that it still meets the requirements.

But isn't this exactly TDD as demonstrated in the video?

No, I don't think so. The TDD demonstration was *test-driven*, but not *requirements driven*. Mapping 1 to "I" and 2 to "II" are tests, but are not true requirements in my view. A good requirement is based on insight into the domain. Just testing that 2 maps to "II" does not provide that insight.

A very simple implementation

After criticizing someone else's implementation, time for me to put up or shut up.

So, what is the simplest implementation I can think of that would work?

How about just converting our arabic number to tally marks? 1 becomes "I", 2 becomes "II", and so on.

```
let arabicToRoman arabic =
   String.replicate arabic "I"
```

Here it is in action:

```
arabicToRoman 1    // "I"
arabicToRoman 5    // "IIIII"
arabicToRoman 10   // "IIIIIIIIII"
```

This code actually meets the first and second requirements already, and for all inputs!

Of course, having 4000 tally marks is not very helpful, which is no doubt why the Romans started abbreviating them.

This is where insight into the domain comes in. If we understand that the tally marks are being abbreviated, we can emulate that in our code.

So let's convert all runs of five tally marks into a "V".

```
let arabicToRoman arabic =
   (String.replicate arabic "I")
    .Replace("IIIII","V")

// test
arabicToRoman 1    // "I"
arabicToRoman 5    // "V"
arabicToRoman 6    // "VI"
arabicToRoman 10   // "VV"
```

But now we can have runs of "V"s. Two "V"s need to be collapsed into an "X".

```
let arabicToRoman arabic =
   (String.replicate arabic "I")
    .Replace("IIIII","V")
    .Replace("VV","X")

// test
arabicToRoman 1    // "I"
arabicToRoman 5    // "V"
arabicToRoman 6    // "VI"
arabicToRoman 10   // "X"
arabicToRoman 12   // "XII"
arabicToRoman 16   // "XVI"
```

I think you get the idea. We can go on adding abbreviations...

```
let arabicToRoman arabic =
   (String.replicate arabic "I")
    .Replace("IIIII","V")
    .Replace("VV","X")
    .Replace("XXXXX","L")
    .Replace("LL","C")
    .Replace("CCCCC","D")
    .Replace("DD","M")

// test
arabicToRoman 1    // "I"
arabicToRoman 5    // "V"
arabicToRoman 6    // "VI"
arabicToRoman 10   // "X"
arabicToRoman 12   // "XII"
arabicToRoman 16   // "XVI"
arabicToRoman 3497 // "MMMCCCCLXXXXVII"
```

And now we're done. We've met the first three requirements.

If we want to add the optional abbreviations for the fours and nines, we can do that at the end, after all the tally marks have been accumulated.

```
let arabicToRoman arabic =
   (String.replicate arabic "I")
    .Replace("IIIII","V")
    .Replace("VV","X")
    .Replace("XXXXX","L")
    .Replace("LL","C")
    .Replace("CCCCC","D")
    .Replace("DD","M")
    // optional substitutions
    .Replace("IIII","IV")
    .Replace("VIV","IX")
    .Replace("XXXX","XL")
    .Replace("LXL","XC")
    .Replace("CCCC","CD")
    .Replace("DCD","CM")


// test
arabicToRoman 1    // "I"
arabicToRoman 4    // "IV"
arabicToRoman 5    // "V"
arabicToRoman 6    // "VI"
arabicToRoman 10   // "X"
arabicToRoman 12   // "XII"
arabicToRoman 16   // "XVI"
arabicToRoman 40   // "XL"
arabicToRoman 946  // "CMXLVI"
arabicToRoman 3497 // "MMMCDXCVII"
```

Here is what I like about this approach:

- It is derived from understanding the domain model (tally marks) rather than jumping right into a recursive design.
- As a result, the implementation follows the requirements very closely. In fact it basically writes itself.
- By following this step-by-step approach, someone else would have high confidence in the code being correct just by examining the code. There is no recursion or special tricks that would confuse anyone.
- The implementation generates output for all inputs at all times. In the intermediate stages, when it doesn't meet all the requirements, it at least generates output (e.g. 10 mapped to "VV") that tells us what we need to do next.

Yes, this might not be the most efficient code, creating strings with 4000 "I"s in them! And of course, a more efficient approach would subtract the large tallies ("M", then "D", then "C") straight from the input, leading to the recursive solution demonstrated in the TDD video.

But on the other hand, this implementation might well be efficient enough. The requirements don't say anything about performance constraints -- YAGNI anyone? -- so I'm tempted to leave it at this.

A bi-quinary coded decimal implementation

I can't resist another implementation, just so that I can use the word "bi-quinary" again.

The implementation will again be based on our understanding of the domain, in this case, the Roman abacus.

In the abacus, each row or wire represents a decimal place, just as our common Arabic notation does. But the number in that place can be encoded by two different symbols, depending on the number.

Some examples:

- 1 in the tens place is encoded by "X"
- 2 in the tens place is encoded by "XX"
- 5 in the tens place is encoded by "L"
- 6 in the tens place is encoded by "LX"

and so on.

This leads directly to an algorithm based on converting the beads on the abacus into a string representation.

- Split the input number into units, tens, hundreds and thousands. These represent each row or wire on the abacus.
- Encode the digit for each place into a string using the "bi-quinary" representation and the appropriate symbols for that place.
- Concat the representations for each place together to make single output string.

Here's an implementation that is a direct translation of that algorithm:

```
let biQuinaryDigits place (unit,five) arabic =
    let digit =  arabic % (10*place) / place
    match digit with
    | 0 -> ""
    | 1 -> unit
    | 2 -> unit + unit
    | 3 -> unit + unit + unit
    | 4 -> unit + unit + unit + unit
    | 5 -> five
    | 6 -> five + unit
    | 7 -> five + unit + unit
    | 8 -> five + unit + unit + unit
    | 9 -> five + unit + unit + unit + unit
    | _ -> failwith "Expected 0-9 only"

let arabicToRoman arabic =
    let units = biQuinaryDigits 1 ("I","V") arabic
    let tens = biQuinaryDigits 10 ("X","L") arabic
    let hundreds = biQuinaryDigits 100 ("C","D") arabic
    let thousands = biQuinaryDigits 1000 ("M","?") arabic
    thousands + hundreds + tens + units
```

Note that the above code does not produce the abbreviations for the four and nine cases. We can easily modify it to do this though. We just need to pass in the symbol for ten, and tweak the mapping for the 4 and 9 case, as follows:

```
let biQuinaryDigits place (unit,five,ten) arabic =
  let digit =  arabic % (10*place) / place
  match digit with
  | 0 -> ""
  | 1 -> unit
  | 2 -> unit + unit
  | 3 -> unit + unit + unit
  | 4 -> unit + five // changed to be one less than five
  | 5 -> five
  | 6 -> five + unit
  | 7 -> five + unit + unit
  | 8 -> five + unit + unit + unit
  | 9 -> unit + ten  // changed to be one less than ten
  | _ -> failwith "Expected 0-9 only"

let arabicToRoman arabic =
  let units = biQuinaryDigits 1 ("I","V","X") arabic
  let tens = biQuinaryDigits 10 ("X","L","C") arabic
  let hundreds = biQuinaryDigits 100 ("C","D","M") arabic
  let thousands = biQuinaryDigits 1000 ("M","?","?") arabic
  thousands + hundreds + tens + units
```

Again, both these implementations are very straightforward and easy to verify. There are no subtle edge cases lurking in the code.

Review

I started off this post being annoyed at a TDD demonstration. Let me review the reasons why I was annoyed, and how my approach differs.

Requirements

The TDD demonstration video did not make any attempt to document the requirements at all. I would say that this a dangerous thing to do, especially if you are learning.

I would prefer that before you start coding you *always* make an effort to be explicit about what you are trying to do.

With only a tiny bit of effort I came up with some explicit requirements that I could use for verification later.

I also explicitly documented the range of valid inputs -- something that was unfortunately lacking in the TDD demonstration.

Understanding the domain

Even if the requirements have been made explicit for you, I think that it is always worthwhile spending time to *really* understand the domain you are working in.

In this case, understanding that Roman numerals were a tally-based system helped with the design later. (Plus I learned what "bi-quinary" means and got to use it in this post!)

Unit tests

The unit tests in the TDD demonstration were built one single case at a time. First zero, then one, and so on.

As I note above, I feel very uncomfortable with this approach because (a) I don't think it leads to a good design and (b) the single cases don't cover all possible inputs.

I would strongly recommend that you write tests that map *directly* to the requirements. If the requirements are any good, this will mean that the tests cover many inputs at once, so you can then test as many inputs as you can.

Ideally, you would use a property-based testing tool like QuickCheck. Not only does it make this approach much easier to implement, but it forces you to identify what the properties of your design should be, which in turn helps you clarify any fuzzy requirements.

Implementation

Finally, I described two implementations, both completely different from the recursive one demonstrated in the TDD video.

Both designs were derived directly from an understanding of the domain. The first from using tally marks, and the second from using an abacus.

To my mind, both of these designs were also easier to understand -- no recursion! -- and thus easier to have confidence in.

Summary

(Added based on comments I made below.)

Let me be clear that I have absolutely no problem with TDD. And I don't have a problem with katas either.

But here's my concern about these kinds of "dive-in" demos, namely that novices and learners might unintentionally learn the following (implicit) lessons:

- It is OK to accept requirements as given without asking questions.
- It is OK to work without a clear idea of the goal.
- It is OK to start coding immediately.
- It is OK to create tests that are extremely specific (e.g. with magic numbers).
- It is OK to consider only the happy path.
- It is OK to do micro refactoring without looking at the bigger picture.

Personally, I think that if you are *practicing* to be a *professional* developer, you should:

- Practice asking for as much information as possible before you start coding.
- Practice writing requirements (from unclear input) in such a way that they can be tested.
- Practice thinking (analyzing and designing) rather than immediately coding.
- Practice creating general tests rather than specific ones.
- Practice thinking about and handling bad inputs, corner cases, and errors.
- Practice major refactoring (rather than micro refactoring) so as to develop an intuition about where the shearing layers should be.

These principles are all completely compatible with TDD (or at least the "London" school of TDD) and programming katas. There is no conflict, and I cannot see why they would be controversial.

What do you think?

I'm sure many of you will disagree with this post. I'm up for a (civilized) debate. Please leave comments below or on Reddit.

If you'd like to see the complete code for this post, it is available as a gist here. The gist also includes full property-based tests for both implementations.

Calculator Walkthrough: Part 1

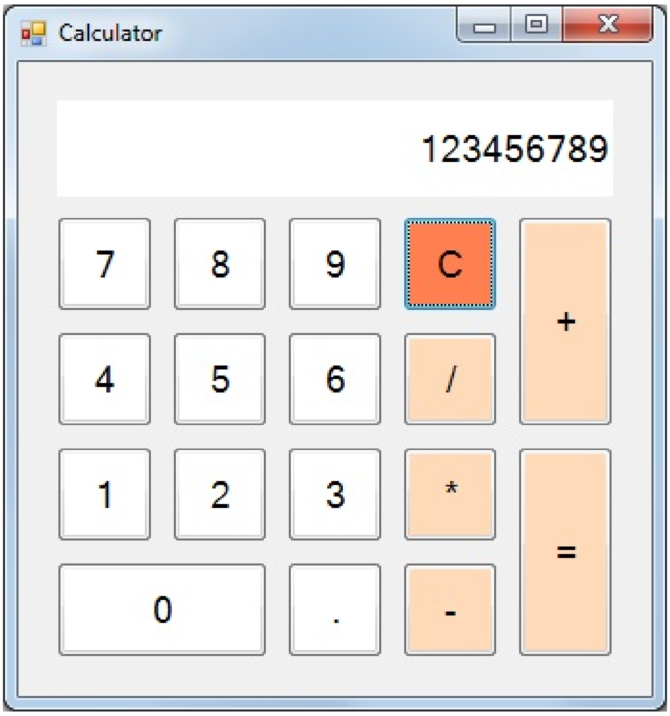
One comment I hear often is a complaint about the gap between theory and practice in F# and functional programming in general. In other words, you know the theory, but how do you actually design and implement an application using FP principles?

So I thought it might be useful to show you how I personally would go about designing and implementing some little applications from beginning to end.

These will be sort of annotated "live coding" sessions. I'll take a problem and start coding it, taking you through my thought process at each stage. I will make mistakes too, so you'll see how I deal with that, and do backtracking and refactoring.

Please be aware that I'm not claiming that this is production ready code. The code I'm going to show you is more like a exploratory sketch, and as a result I will do certain bad things (like not testing!) which I would not do for more critical code.

For this first post in the series, I'll be developing a simple pocket calculator app, like this:



My development approach

My approach to software development is eclectic and pragmatic -- I like to mix different techniques and alternate between top-down and bottom-up approaches.

Typically I start with the requirements -- I'm a fan of requirements-driven design! Ideally, I would aim to become an expert in the domain as well.

Next, I work on modelling the domain, using domain-driven design with a focus on domain events ("event storming"), not just static data ("aggregates" in DDD terminology).

As part of the modelling process, I sketch a design using type-first development to create types that represent both the domain data types ("nouns") and the domain activities ("verbs").

After doing a first draft of the domain model, I typically switch to a "bottom up" approach and code a small prototype that exercises the model that I have defined so far.

Doing some real coding at this point acts as a reality check. It ensures that the domain model actually makes sense and is not too abstract. And of course, it often drives more questions about the requirements and domain model, so I go back to step 1, do some refining and refactoring, and rinse and repeat until I am happy.

(Now if I was working with a team on a large project, at this point we could also start building a real system incrementally and start on the user interface (e.g. with paper prototypes). Both of these activities will typically generate yet more questions and changes in requirements too, so the whole process is cyclical at all levels.)

So this would be my approach in a perfect world. In practice, of course, the world is not perfect. There is bad management to contend with, a lack of requirements, silly deadlines and more, all of which mean that I rarely get to use an ideal process.

But in this example, I'm the boss, so if I don't like the result, I've only myself to blame!

Getting started

So, let's get started. What should we do first?

Normally I would start with requirements. But do I *really* need to spend a lot of time writing up requirements for a calculator?

I'm going to be lazy and say no. Instead I'm just to dive in -- I'm confident that I know how a calculator works. (*As you'll see later, I was wrong! Trying to write up the requirements would have been a good exercise, as there are some interesting edge cases.*)

So let's start with the type-first design instead.

In my designs, every use-case is a function, with one input and one output.

For this example then, we need to model the public interface to the Calculator as a function. Here's the signature:

```
type Calculate = CalculatorInput -> CalculatorOutput
```

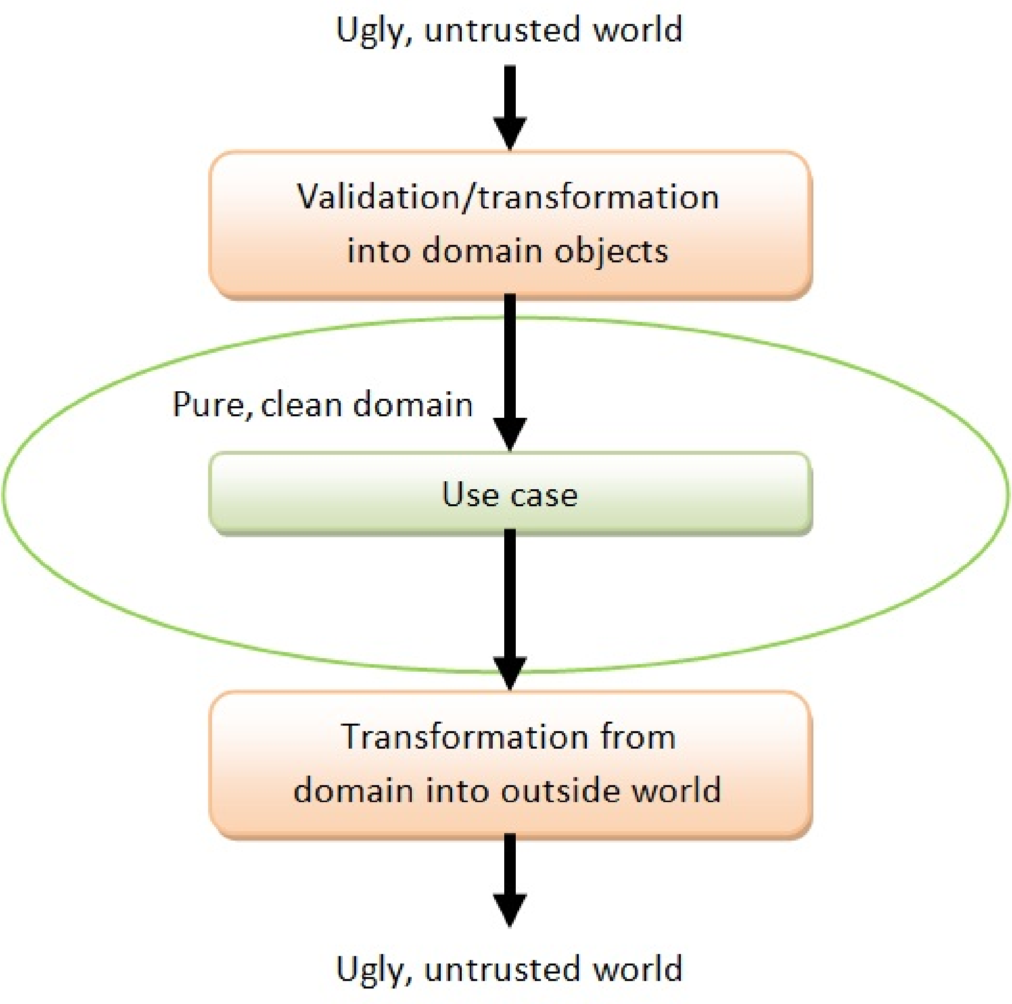
That was easy! The first question then is: are there any other use-cases that we need to model? I think for now, no. We'll just start with a single case that handles all the inputs.

Defining the input and output to the function

But now we have created two new types, `CalculatorInput` and `CalculatorOutput`, that are undefined (and if you type this into a F# script file, you'll have red squigglies to remind you). We'd better define those now.

Before moving on, I should make it very clear that the input and output types for this function are going to be pure and clean. When designing our domain we never want to be dealing with the messy world of strings, primitive datatypes, validation, and so on.

Instead there will typically be a validation/transformation function that converts from the messy untrusted world into our lovely, pristine domain on the way in, and another similar function that does the reverse on the way out.



Ok, let's work on the `CalculatorInput` first. What would the structure of the input look like?

First, obviously, there will be some keystrokes, or some other way of communicating the intent of the user. But also, since the calculator is stateless, we need to pass in some state as well. This state would contain, for example, the digits typed in so far.

As to the output, the function will have to emit a new, updated state, of course.

But do we need anything else, such as a structure containing formatted output for display? I don't think we do. We want to isolate ourselves from the display logic, so we'll just let the UI turn the state into something that can be displayed.

What about errors? In other posts, I have spent a lot of time talking about error handling. Is it needed in this case?

In this case, I think not. In a cheap pocket calculator, any errors are shown right in the display, so we'll stick with that approach for now.

So here's the new version of the function:

```
type Calculate = CalculatorInput * CalculatorState -> CalculatorState
```

`CalculatorInput` now means the keystrokes or whatever, and `CalculatorState` is the state.

Notice that I have defined this function using a tuple (`CalculatorInput * CalculatorState`) as input, rather than as two separate parameters (which would look like `CalculatorInput -> CalculatorState -> CalculatorState`). I did this because both parameters are always needed and a tuple makes this clear -- I don't want to be partially applying the input, for example.

In fact I do this for all functions when doing type-first design. Every function has one input and one output. This doesn't mean that there might not be potential for doing partial application later, just that, at the design stage, I only want one parameter.

Also note that things that are not part of the pure domain (such as configuration and connection strings) will *never* be shown at this stage, although, at implementation time, they will of course be added to the functions that implement the design.

Defining the CalculatorState type

Now let's look at the `CalculatorState` . All I can think of that we need right now is something to hold the information to display.

```
type Calculate = CalculatorInput * CalculatorState -> CalculatorState
and CalculatorState = {
    display: CalculatorDisplay
    }
```

I've defined a type `CalculatorDisplay` , firstly as documentation to make it clear what the field value is used for, and secondly, so I can postpone deciding what the display actually is!

So what should the type of the display be? A float? A string? A list of characters? A record with multiple fields?

Well, I'm going to go for `string` , because, as I said above, we might need to display errors.

```
type Calculate = CalculatorInput * CalculatorState -> CalculatorState
and CalculatorState = {
    display: CalculatorDisplay
    }
and CalculatorDisplay = string
```

Notice that I am using `and` to connect the type definitions together. Why?

Well, F# compiles from top to bottom, so you must define a type before it is used. The following code will not compile:

```
type Calculate = CalculatorInput * CalculatorState -> CalculatorState
type CalculatorState = {
    display: CalculatorDisplay
    }
type CalculatorDisplay = string
```

I could fix this by changing the order of the declarations, but since I am in "sketch" mode, and I don't want to reorder things all the time, I will just append new declarations to the bottom and use `and` to connect them.

In the final production code though, when the design has stabilized, I *would* reorder these types to avoid using `and`. The reason is that `and` can hide cycles between types and prevent refactoring.

Defining the CalculatorInput type

For the `CalculatorInput` type, I'll just list all the buttons on the calculator!

```
// as above
and CalculatorInput =
    | Zero | One | Two | Three | Four
    | Five | Six | Seven | Eight | Nine
    | DecimalSeparator
    | Add | Subtract | Multiply | Divide
    | Equals | Clear
```

Some people might say: why not use a `char` as the input? But as I explained above, in my domain I only want to deal with ideal data. By using a limited set of choices like this, I never have to deal with unexpected input.

Also, a side benefit of using abstract types rather than chars is that `DecimalSeparator` is not assumed to be ".". The actual separator should be obtained by first getting the current culture (`System.Globalization.CultureInfo.CurrentCulture`) and then using `CurrentCulture.NumberFormat.CurrencyDecimalSeparator` to get the separator. By hiding this implementation detail from the design, changing the actual separator used will have minimal effect on the code.

Refining the design: handling digits

So that's a first pass at the design done. Now let's dig deeper and define some of the internal processes.

Let's start with how the digits are handled.

When a digit key is pressed, we want to append the digit to the current display. Let's define a function type that represents that:

```
type UpdateDisplayFromDigit = CalculatorDigit * CalculatorDisplay -> CalculatorDisplay
```

The `CalculatorDisplay` type is the one we defined earlier, but what is this new `CalculatorDigit` type?

Well obviously we need some type to represent all the possible digits that can be used as input. Other inputs, such as `Add` and `Clear`, would not be valid for this function.

```
type CalculatorDigit =
    | Zero | One | Two | Three | Four
    | Five | Six | Seven | Eight | Nine
    | DecimalSeparator
```

So the next question is, how do we get a value of this type? Do we need a function that maps a `CalculatorInput` to a `CalculatorDigit` type, like this?

```
let convertInputToDigit (input:CalculatorInput) =
    match input with
        | Zero -> CalculatorDigit.Zero
        | One -> CalculatorDigit.One
        | etc
        | Add -> ???
        | Clear -> ???
```

In many situations, this might be necessary, but in this case it seems like overkill. And also, how would this function deal with non-digits such as `Add` and `Clear`?

So let's just redefine the `CalculatorInput` type to use the new type directly:

```
type CalculatorInput =
    | Digit of CalculatorDigit
    | Add | Subtract | Multiply | Divide
    | Equals | Clear
```

While we're at it, let's classify the other buttons as well.

I would classify `Add | Subtract | Multiply | Divide` as math operations, and as for `Equals | Clear`, I'll just call them "actions" for lack of better word.

Here's the complete refactored design with new types `CalculatorDigit` , `CalculatorMathOp` and `CalculatorAction` :

```
type Calculate = CalculatorInput * CalculatorState -> CalculatorState
and CalculatorState = {
    display: CalculatorDisplay
    }
and CalculatorDisplay = string
and CalculatorInput =
    | Digit of CalculatorDigit
    | Op of CalculatorMathOp
    | Action of CalculatorAction
and CalculatorDigit =
    | Zero | One | Two | Three | Four
    | Five | Six | Seven | Eight | Nine
    | DecimalSeparator
and CalculatorMathOp =
    | Add | Subtract | Multiply | Divide
and CalculatorAction =
    | Equals | Clear

type UpdateDisplayFromDigit = CalculatorDigit * CalculatorDisplay -> CalculatorDisplay
```

This is not the only approach. I could have easily left `Equals` and `Clear` as separate choices.

Now let's revisit `UpdateDisplayFromDigit` again. Do we need any other parameters? For example, do we need any other part of the state?

No, I can't think of anything else. When defining these functions, I want to be as minimal as possible. Why pass in the whole calculator state if you only need the display?

Also, would `UpdateDisplayFromDigit` ever return an error? For example, surely we can't add digits indefinitely -- what happens when we are not allowed to? And is there some other combination of inputs that might cause an error? For example, inputting nothing but decimal separators! What happens then?

For this little project, I will assume that neither of these will create an explicit error, but instead, bad input will be rejected silently. In other words, after 10 digits, say, other digits will be ignored. And after the first decimal separator, subsequent ones will be ignored as well.

Alas, I cannot encode these requirements in the design. But that fact that `UpdateDisplayFromDigit` does not return any explicit error type *does* at least tell me that errors will be handled silently.

Refining the design: the math operations

Now let's move on to the math operations.

These are all binary operations, taking two numbers and spitting out a new result.

A function type to represent this would look like this:

```
type DoMathOperation = CalculatorMathOp * Number * Number -> Number
```

If there were unary operations as well, such as `1/x`, we would need a different type for those, but we don't, so we can keep things simple.

Next decision: what numeric type should we use? Should we make it generic?

Again, let's just keep it simple and use `float`. But we'll keep the `Number` alias around to decouple the representation a bit. Here's the updated code:

```
type DoMathOperation = CalculatorMathOp * Number * Number -> Number
and Number = float
```

Now let's ponder `DoMathOperation`, just as we did for `UpdateDisplayFromDigit` above.

Question 1: Is this the minimal set of parameters? For example, do we need any other part of the state?

Answer: No, I can't think of anything else.

Question 2: Can `DoMathOperation` ever return an error?

Answer: Yes! What about dividing by zero?

So how should we handle errors? Let's create a new type that represents a result of a math operation, and make that the output of `DoMathOperation`:

The new type, `MathOperationResult` will have two choices (discriminated union) between `Success` and `Failure`.

```
type DoMathOperation = CalculatorMathOp * Number * Number -> MathOperationResult
and Number = float
and MathOperationResult =
    | Success of Number
    | Failure of MathOperationError
and MathOperationError =
    | DivideByZero
```

We could have also used the built-in generic `Choice` type, or even a full "railway oriented programming" approach, but since this is a sketch of the design, I want the design to stand alone, without a lot of dependencies, so I'll just define the specific type right here.

Any other errors? NaNs or underflows or overflows? I'm not sure. We have the `MathOperationError` type, and it would be easy to extend it as needed.

Where do numbers come from?

We've defined `DoMathOperation` to use `Number` values as input. But where does a `Number` come from?

Well they come from the sequence of digits that have been entered -- converting the digits into a float.

One approach would be to store a `Number` in the state along with the string display, and update it as each digit comes in.

I'm going to take a simpler approach, and just get the number from the display directly. In other words, we need a function that looks like this:

```
type GetDisplayNumber = CalculatorDisplay -> Number
```

Thinking about it though, the function could fail, because the display string could be "error" or something. So let's return an option instead.

```
type GetDisplayNumber = CalculatorDisplay -> Number option
```

Similarly, when we *do* have a successful result, we will want to display it, so we need a function that works in the other direction:

```
type SetDisplayNumber = Number -> CalculatorDisplay
```

This function can never error (I hope), so we don't need the `option`.

Refining the design: handling a math operation input

We're not done with math operations yet, though!

What is the visible effect when the input is `Add` ? None!

The `Add` event needs another number to be entered later, so the `Add` event is somehow kept pending, waiting for the next number.

If you think about, we not only have to keep the `Add` event pending, but also the previous number, ready to be added to the latest number that is input.

Where will we keep track of this? In the `CalculatorState` of course!

Here's our first attempt to add the new fields:

```
and CalculatorState = {
    display: CalculatorDisplay
    pendingOp: CalculatorMathOp
    pendingNumber: Number
    }
```

But sometimes there isn't a pending operation, so we have to make it optional:

```
and CalculatorState = {
    display: CalculatorDisplay
    pendingOp: CalculatorMathOp option
    pendingNumber: Number option
    }
```

But this is wrong too! Can we have a `pendingOp` without a `pendingNumber` , or vice versa? No. They live and die together.

This implies that the state should contain a pair, and the whole pair is optional, like this:

```
and CalculatorState = {
    display: CalculatorDisplay
    pendingOp: (CalculatorMathOp * Number) option
    }
```

But now we are still missing a piece. If the operation is added to the state as pending, when does the operation actually get *run* and the result displayed?

Answer: when the `Equals` button is pushed, or indeed any another math op button. We'll deal with that later.

Refining the design: handling the Clear button

We've got one more button to handle, the `Clear` button. What does it do?

Well, it obviously just resets the state so that the display is empty and any pending operations are removed.

I'm going to call this function `InitState` rather than "clear", and here is its signature:

```
type InitState = unit -> CalculatorState
```

Defining the services

At this point, we have everything we need to switch to bottom up development. I'm eager to try building a trial implementation of the `Calculate` function, to see if the design is usable, and if we've missed anything.

But how can I create a trial implementation without implementing the whole thing?

This is where all these types come in handy. We can define a set of "services" that the `calculate` function will use, but without actually implementing them!

Here's what I mean:

```
type CalculatorServices = {
    updateDisplayFromDigit: UpdateDisplayFromDigit
    doMathOperation: DoMathOperation
    getDisplayNumber: GetDisplayNumber
    setDisplayNumber: SetDisplayNumber
    initState: InitState
    }
```

We've created a set of services that can be injected into an implementation of the `Calculate` function. With these in place, we can code the `Calculate` function immediately and deal with the implementation of the services later.

At this point, you might be thinking that this seems like overkill for a tiny project.

It's true -- we don't want this to turn into FizzBuzz Enterprise Edition!

But I'm demonstrating a principle here. By separating the "services" from the core code, you can start prototyping immediately. The goal is not to make a production ready codebase, but to find any issues in the design. We are still in the requirements discovery phase.

This approach should not be unfamiliar to you -- it is directly equivalent to the OO principle of creating a bunch of interfaces for services and then injecting them into the core domain.

Review

So let's review -- with the addition of the services, our initial design is complete. Here is all the code so far:

```
type Calculate = CalculatorInput * CalculatorState -> CalculatorState
and CalculatorState = {
    display: CalculatorDisplay
    pendingOp: (CalculatorMathOp * Number) option
    }
and CalculatorDisplay = string
and CalculatorInput =
    | Digit of CalculatorDigit
    | Op of CalculatorMathOp
    | Action of CalculatorAction
and CalculatorDigit =
    | Zero | One | Two | Three | Four
    | Five | Six | Seven | Eight | Nine
    | DecimalSeparator
and CalculatorMathOp =
    | Add | Subtract | Multiply | Divide
and CalculatorAction =
    | Equals | Clear
and UpdateDisplayFromDigit =
    CalculatorDigit * CalculatorDisplay -> CalculatorDisplay
and DoMathOperation =
    CalculatorMathOp * Number * Number -> MathOperationResult
and Number = float
and MathOperationResult =
    | Success of Number
    | Failure of MathOperationError
and MathOperationError =
    | DivideByZero

type GetDisplayNumber =
    CalculatorDisplay -> Number option
type SetDisplayNumber =
    Number -> CalculatorDisplay

type InitState =
    unit -> CalculatorState

type CalculatorServices = {
    updateDisplayFromDigit: UpdateDisplayFromDigit
    doMathOperation: DoMathOperation
    getDisplayNumber: GetDisplayNumber
    setDisplayNumber: SetDisplayNumber
    initState: InitState
    }
```

Summary

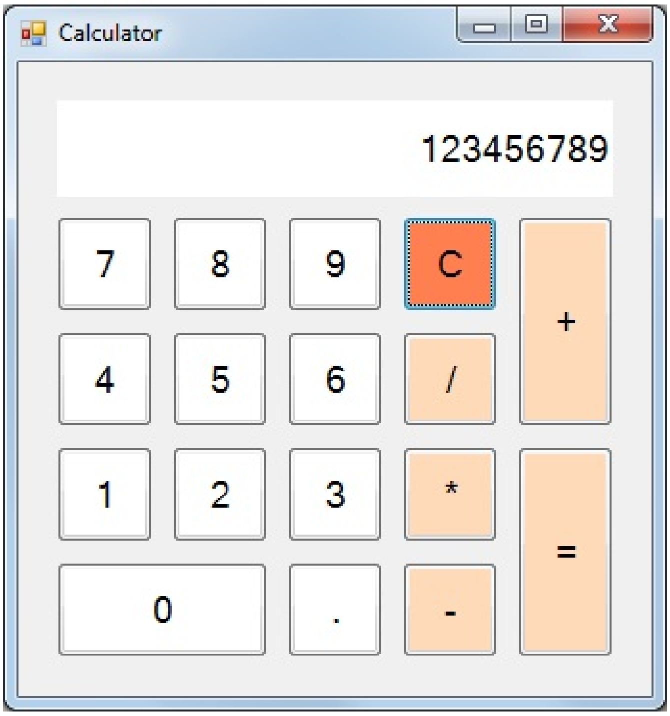
I think that this is quite nice. We haven't written any "real" code yet, but with a bit of thought, we have already built quite a detailed design.

In the next post, I'll put this design to the test by attempting to create an implementation.

The code for this post is available in this gist on GitHub.

Calculator Walkthrough: Part 2

In this post, I'll continue developing a simple pocket calculator app, like this:



In the previous post, we completed a first draft of the design, using only types (no UML diagrams!).

Now it's time to create a trial implementation that uses the design.

Doing some real coding at this point acts as a reality check. It ensures that the domain model actually makes sense and is not too abstract. And of course, it often drives more questions about the requirements and domain model.

First implementation

So let's try implementing the main calculator function, and see how we do.

First, we can immediately create a skeleton that matches each kind of input and processes it accordingly.

```
let createCalculate (services:CalculatorServices) :Calculate =
    fun (input,state) ->
        match input with
        | Digit d ->
            let newState = // do something
            newState //return
        | Op op ->
            let newState = // do something
            newState //return
        | Action Clear ->
            let newState = // do something
            newState //return
        | Action Equals ->
            let newState = // do something
            newState //return
```

You can see that this skeleton has a case for each type of input to handle it appropriately. Note that in all cases, a new state is returned.

This style of writing a function might look strange though. Let's look at it a bit more closely.

First, we can see that `createCalculate` is the not the calculator function itself, but a function that *returns* another function. The returned function is a value of type `Calculate` -- that's what the `:Calculate` at the end means.

Here's just the top part:

```
let createCalculate (services:CalculatorServices) :Calculate =
    fun (input,state) ->
        match input with
            // code
```

Since it is returning a function, I chose to write it using a lambda. That's what the `fun (input,state) ->` is for.

But I could have also written it using an inner function, like this

```
let createCalculate (services:CalculatorServices) :Calculate =
    let innerCalculate (input,state) =
        match input with
            // code
    innerCalculate // return the inner function
```

Both approaches are basically the same* -- take your pick!

* Although there might be some performance differences.

Dependency injection of services

But `createCalculate` doesn't just return a function, it also has a `services` parameter. This parameter is used for doing the "dependency injection" of the services.

That is, the services are only used in `createCalculate` itself, and are not visible in the function of type `Calculate` that is returned.

The "main" or "bootstrapper" code that assembles all the components for the application would look something like this:

```
// create the services
let services = CalculatorServices.createServices()

// inject the services into the "factory" method
let calculate = CalculatorImplementation.createCalculate services

// the returned "calculate" function is of type Calculate
// and can be passed into the UI, for example

// create the UI and run it
let form = new CalculatorUI.CalculatorForm(calculate)
form.Show()
```

Implementation: handling digits

Now let's start implementing the various parts of the calculation function. We'll start with the digits handling logic.

To keep the main function clean, let's pass the reponsibility for all the work to a helper function `updateDisplayFromDigit` , like this:

```
let createCalculate (services:CalculatorServices) :Calculate =
    fun (input,state) ->
        match input with
        | Digit d ->
            let newState = updateDisplayFromDigit services d state
            newState //return
```

Note that I'm creating a `newState` value from the result of `updateDisplayFromDigit` and then returning it as a separate step.

I could have done the same thing in one step, without an explicit `newState` value, as shown below:

```
let createCalculate (services:CalculatorServices) :Calculate =
    fun (input,state) ->
        match input with
        | Digit d ->
            updateDisplayFromDigit services d state
```

Neither approach is automatically best. I would pick one or the other depending on the context.

For simple cases, I would avoid the extra line as being unnecessary, but sometimes having an explicit return value is more readable. The name of the value tells you an indication of the return type, and it gives you something to watch in the debugger, if you need to.

Alright, let's implement `updateDisplayFromDigit` now. It's pretty straightforward.

- first use the `updateDisplayFromDigit` in the services to actually update the display
- then create a new state from the new display and return it.

```
let updateDisplayFromDigit services digit state =
    let newDisplay = services.updateDisplayFromDigit (digit,state.display)
    let newState = {state with display=newDisplay}
    newState //return
```

Implementation: handling Clear and Equals

Before we move onto the implementation of the math operations, lets look at handling `Clear` and `Equals`, as they are simpler.

For `Clear`, just init the state, using the provided `initState` service.

For `Equals`, we check if there is a pending math op. If there is, run it and update the display, otherwise do nothing. We'll put that logic in a helper function called `updateDisplayFromPendingOp`.

So here's what `createCalculate` looks like now:

```
let createCalculate (services:CalculatorServices) :Calculate =
    fun (input,state) ->
        match input with
        | Digit d -> // as above
        | Op op -> // to do
        | Action Clear ->
            let newState = services.initState()
            newState //return
        | Action Equals ->
            let newState = updateDisplayFromPendingOp services state
            newState //return
```

Now to `updateDisplayFromPendingOp`. I spent a few minutes thinking about, and I've come up with the following algorithm for updating the display:

- First, check if there is any pending op. If not, then do nothing.
- Next, try to get the current number from the display. If you can't, then do nothing.
- Next, run the op with the pending number and the current number from the display. If you get an error, then do nothing.
- Finally, update the display with the result and return a new state.
- The new state also has the pending op set to `None`, as it has been processed.

And here's what that logic looks like in imperative style code:

```
// First version of updateDisplayFromPendingOp
// * very imperative and ugly
let updateDisplayFromPendingOp services state =
    if state.pendingOp.IsSome then
        let op,pendingNumber = state.pendingOp.Value
        let currentNumberOpt = services.getDisplayNumber state.display
        if currentNumberOpt.IsSome then
            let currentNumber = currentNumberOpt.Value
            let result = services.doMathOperation (op,pendingNumber,currentNumber)
            match result with
            | Success resultNumber ->
                let newDisplay = services.setDisplayNumber resultNumber
                let newState = {display=newDisplay; pendingOp=None}
                newState //return
            | Failure error ->
                state // original state is untouched
        else
            state // original state is untouched
    else
        state // original state is untouched
```

Ewww! Don't try that at home!

That code does follow the algorithm exactly, but is really ugly and also error prone (using `.Value` on an option is a code smell).

On the plus side, we did make extensive use of our "services", which has isolated us from the actual implementation details.

So, how can we rewrite it to be more functional?

Bumping into bind

The trick is to recognize that the pattern "if something exists, then act on that value" is exactly the `bind` pattern discussed here and here.

In order to use the bind pattern effectively, it's a good idea to break the code into many small chunks.

First, the code `if state.pendingOp.IsSome then do something` can be replaced by `Option.bind`.

```
let updateDisplayFromPendingOp services state =
    let result =
        state.pendingOp
        |> Option.bind ???
```

But remember that the function has to return a state. If the overall result of the bind is `None`, then we have *not* created a new state, and we must return the original state that was passed in.

This can be done with the built-in `defaultArg` function which, when applied to an option, returns the option's value if present, or the second parameter if `None`.

```
let updateDisplayFromPendingOp services state =
    let result =
        state.pendingOp
        |> Option.bind ???
    defaultArg result state
```

You can also tidy this up a bit as well by piping the result directly into `defaultArg`, like this:

```
let updateDisplayFromPendingOp services state =
    state.pendingOp
    |> Option.bind ???
    |> defaultArg <| state
```

I admit that the reverse pipe for `state` looks strange -- it's definitely an acquired taste!

Onwards! Now what about the parameter to `bind`? When this is called, we know that pendingOp is present, so we can write a lambda with those parameters, like this:

```
let result =
    state.pendingOp
    |> Option.bind (fun (op,pendingNumber) ->
        let currentNumberOpt = services.getDisplayNumber state.display
        // code
        )
```

Alternatively, we could create a local helper function instead, and connect it to the bind, like this:

```
let executeOp (op,pendingNumber) =
    let currentNumberOpt = services.getDisplayNumber state.display
    /// etc

let result =
    state.pendingOp
    |> Option.bind executeOp
```

I myself generally prefer the second approach when the logic is complicated, as it allows a chain of binds to be simple. That is, I try to make my code look like:

```
let doSomething input = return an output option
let doSomethingElse input = return an output option
let doAThirdThing input = return an output option

state.pendingOp
|> Option.bind doSomething
|> Option.bind doSomethingElse
|> Option.bind doAThirdThing
```

Note that in this approach, each helper function has a non-option for input but always must output an *option*.

Using bind in practice

Once we have the pending op, the next step is to get the current number from the display so we can do the addition (or whatever).

Rather than having a lot of logic, I'm going keep the helper function (`getCurrentNumber`) simple.

- The input is the pair (op,pendingNumber)
- The output is the triple (op,pendingNumber,currentNumber) if currentNumber is `Some` , otherwise `None` .

In other words, the signature of `getCurrentNumber` will be `pair -> triple option` , so we can be sure that is usable with the `Option.bind` function.

How to convert the pair into the triple? This can be done just by using `Option.map` to convert the currentNumber option to a triple option. If the currentNumber is `Some` , then the output of the map is `Some triple` . On the other hand, if the currentNumber is `None` , then the output of the map is `None` also.

```
let getCurrentNumber (op,pendingNumber) =
    let currentNumberOpt = services.getDisplayNumber state.display
    currentNumberOpt
    |> Option.map (fun currentNumber -> (op,pendingNumber,currentNumber))

let result =
    state.pendingOp
    |> Option.bind getCurrentNumber
    |> Option.bind ???
```

We can rewrite `getCurrentNumber` to be a bit more idiomatic by using pipes:

```
let getCurrentNumber (op,pendingNumber) =
    state.display
    |> services.getDisplayNumber
    |> Option.map (fun currentNumber -> (op,pendingNumber,currentNumber))
```

Now that we have a triple with valid values, we have everything we need to write a helper function for the math operation.

- It takes a triple as input (the output of `getCurrentNumber`)
- It does the math operation
- It then pattern matches the Success/Failure result and outputs the new state if applicable.

```
let doMathOp (op,pendingNumber,currentNumber) =
    let result = services.doMathOperation (op,pendingNumber,currentNumber)
    match result with
    | Success resultNumber ->
        let newDisplay = services.setDisplayNumber resultNumber
        let newState = {display=newDisplay; pendingOp=None}
        Some newState //return something
    | Failure error ->
        None // failed
```

Note that, unlike the earlier version with nested ifs, this version returns `Some` on success and `None` on failure.

Displaying errors

Writing the code for the `Failure` case made me realize something. If there is a failure, we are not displaying it *at all*, just leaving the display alone. Shouldn't we show an error or something?

Hey, we just found a requirement that got overlooked! This is why I like to create an implementation of the design as soon as possible. Writing real code that deals with all the cases will invariably trigger a few "what happens in this case?" moments.

So how are we going to implement this new requirement?

In order to do this, we'll need a new "service" that accepts a `MathOperationError` and generates a `CalculatorDisplay`.

```
type SetDisplayError = MathOperationError -> CalculatorDisplay
```

and we'll need to add it to the `CalculatorServices` structure too:

```
type CalculatorServices = {
    // as before
    setDisplayNumber: SetDisplayNumber
    setDisplayError: SetDisplayError
    initState: InitState
    }
```

`doMathOp` can now be altered to use the new service. Both `Success` and `Failure` cases now result in a new display, which in turn is wrapped in a new state.

```
let doMathOp (op,pendingNumber,currentNumber) =
    let result = services.doMathOperation (op,pendingNumber,currentNumber)
    let newDisplay =
        match result with
        | Success resultNumber ->
            services.setDisplayNumber resultNumber
        | Failure error ->
            services.setDisplayError error
    let newState = {display=newDisplay;pendingOp=None}
    Some newState //return something
```

I'm going to leave the `Some` in the result, so we can stay with `Option.bind` in the result pipeline*.

* An alternative would be to not return `Some`, and then use `Option.map` in the result pipeline

Putting it all together, we have the final version of `updateDisplayFromPendingOp`. Note that I've also added a `ifNone` helper that makes defaultArg better for piping.

```
// helper to make defaultArg better for piping
let ifNone defaultValue input =
    // just reverse the parameters!
    defaultArg input defaultValue

// Third version of updateDisplayFromPendingOp
// * Updated to show errors on display in Failure case
// * replaces awkward defaultArg syntax
let updateDisplayFromPendingOp services state =
    // helper to extract CurrentNumber
    let getCurrentNumber (op,pendingNumber) =
        state.display
        |> services.getDisplayNumber
        |> Option.map (fun currentNumber -> (op,pendingNumber,currentNumber))

    // helper to do the math op
    let doMathOp (op,pendingNumber,currentNumber) =
        let result = services.doMathOperation (op,pendingNumber,currentNumber)
        let newDisplay =
            match result with
            | Success resultNumber ->
                services.setDisplayNumber resultNumber
            | Failure error ->
                services.setDisplayError error
        let newState = {display=newDisplay;pendingOp=None}
        Some newState //return something

    // connect all the helpers
    state.pendingOp
    |> Option.bind getCurrentNumber
    |> Option.bind doMathOp
    |> ifNone state // return original state if anything fails
```

Using a "maybe" computation expression instead of bind

So far, we've being using "bind" directly. That has helped by removing the cascading `if/else`.

But F# allows you to hide the complexity in a different way, by creating computation expressions.

Since we are dealing with Options, we can create a "maybe" computation expression that allows clean handling of options. (If we were dealing with other types, we would need to create a different computation expression for each type).

Here's the definition -- only four lines!

```
type MaybeBuilder() =
    member this.Bind(x, f) = Option.bind f x
    member this.Return(x) = Some x

let maybe = new MaybeBuilder()
```

With this computation expression available, we can use `maybe` instead of bind, and our code would look something like this:

```
let doSomething input = return an output option
let doSomethingElse input = return an output option
let doAThirdThing input = return an output option

let finalResult = maybe {
    let! result1 = doSomething
    let! result2 = doSomethingElse result1
    let! result3 = doAThirdThing result2
    return result3
    }
```

In our case, then we can write yet another version of `updateDisplayFromPendingOp` -- our fourth!

```
// Fourth version of updateDisplayFromPendingOp
// * Changed to use "maybe" computation expression
let updateDisplayFromPendingOp services state =

    // helper to do the math op
    let doMathOp (op,pendingNumber,currentNumber) =
        let result = services.doMathOperation (op,pendingNumber,currentNumber)
        let newDisplay =
            match result with
            | Success resultNumber ->
                services.setDisplayNumber resultNumber
            | Failure error ->
                services.setDisplayError error
        {display=newDisplay;pendingOp=None}

    // fetch the two options and combine them
    let newState = maybe {
        let! (op,pendingNumber) = state.pendingOp
        let! currentNumber = services.getDisplayNumber state.display
        return doMathOp (op,pendingNumber,currentNumber)
        }
    newState |> ifNone state
```

Note that in *this* implementation, I don't need the `getCurrentNumber` helper any more, as I can just call `services.getDisplayNumber` directly.

So, which of these variants do I prefer?

It depends.

- If there is a very strong "pipeline" feel, as in the ROP approach, then I prefer using an explicit `bind`.
- On the other hand, if I am pulling options from many different places, and I want to combine them in various ways, the `maybe` computation expression makes it easier.

So, in this case, I'll go for the last implementation, using `maybe`.

Implementation: handling math operations

Now we are ready to do the implementation of the math operation case.

First, if there is a pending operation, the result will be shown on the display, just as for the `Equals` case. But *in addition*, we need to push the new pending operation onto the state as well.

For the math operation case, then, there will be *two* state transformations, and `createCalculate` will look like this:

```
let createCalculate (services:CalculatorServices) :Calculate =
    fun (input,state) ->
        match input with
        | Digit d -> // as above
        | Op op ->
            let newState1 = updateDisplayFromPendingOp services state
            let newState2 = addPendingMathOp services op newState1
            newState2 //return
```

We've already defined `updateDisplayFromPendingOp` above. So we just need `addPendingMathOp` as a helper function to push the operation onto the state.

The algorithm for `addPendingMathOp` is:

- Try to get the current number from the display. If you can't, then do nothing.
- Update the state with the op and current number.

Here's the ugly version:

```
// First version of addPendingMathOp
// * very imperative and ugly
let addPendingMathOp services op state =
    let currentNumberOpt = services.getDisplayNumber state.display
    if currentNumberOpt.IsSome then
        let currentNumber = currentNumberOpt.Value
        let pendingOp = Some (op,currentNumber)
        let newState = {state with pendingOp=pendingOp}
        newState //return
    else
        state // original state is untouched
```

Again, we can make this more functional using exactly the same techniques we used for `updateDisplayFromPendingOp` .

So here's the more idiomatic version using `Option.map` and a `newStateWithPending` helper function:

```
// Second version of addPendingMathOp
// * Uses "map" and helper function
let addPendingMathOp services op state =
    let newStateWithPending currentNumber =
        let pendingOp = Some (op,currentNumber)
        {state with pendingOp=pendingOp}

    state.display
    |> services.getDisplayNumber
    |> Option.map newStateWithPending
    |> ifNone state
```

And here's one using `maybe` :

```
// Third version of addPendingMathOp
// * Uses "maybe"
let addPendingMathOp services op state =
    maybe {
        let! currentNumber =
            state.display |> services.getDisplayNumber
        let pendingOp = Some (op,currentNumber)
        return {state with pendingOp=pendingOp}
        }
    |> ifNone state // return original state if anything fails
```

As before, I'd probably go for the last implementation using `maybe` . But the `Option.map` one is fine too.

Implementation: review

Now we're done with the implementation part. Let's review the code:

```
let updateDisplayFromDigit services digit state =
    let newDisplay = services.updateDisplayFromDigit (digit,state.display)
    let newState = {state with display=newDisplay}
    newState //return

let updateDisplayFromPendingOp services state =

    // helper to do the math op
    let doMathOp (op,pendingNumber,currentNumber) =
        let result = services.doMathOperation (op,pendingNumber,currentNumber)
        let newDisplay =
            match result with
            | Success resultNumber ->
                services.setDisplayNumber resultNumber
            | Failure error ->
                services.setDisplayError error
        {display=newDisplay;pendingOp=None}

    // fetch the two options and combine them
    let newState = maybe {
        let! (op,pendingNumber) = state.pendingOp
        let! currentNumber = services.getDisplayNumber state.display
        return doMathOp (op,pendingNumber,currentNumber)
        }
    newState |> ifNone state

let addPendingMathOp services op state =
    maybe {
        let! currentNumber =
            state.display |> services.getDisplayNumber
        let pendingOp = Some (op,currentNumber)
        return {state with pendingOp=pendingOp}
        }
    |> ifNone state // return original state if anything fails

let createCalculate (services:CalculatorServices) :Calculate =
    fun (input,state) ->
        match input with
        | Digit d ->
            let newState = updateDisplayFromDigit services d state
            newState //return
        | Op op ->
            let newState1 = updateDisplayFromPendingOp services state
            let newState2 = addPendingMathOp services op newState1
            newState2 //return
        | Action Clear ->
            let newState = services.initState()
            newState //return
        | Action Equals ->
            let newState = updateDisplayFromPendingOp services state
            newState //return
```

Not bad -- the whole implementation is less than 60 lines of code.

Summary

We have proved that our design is reasonable by making an implementation -- plus we found a missed requirement.

In the next post, we'll implement the services and the user interface to create a complete application.

The code for this post is available in this gist on GitHub.

Calculator Walkthrough: Part 3

In this post, I'll continue developing a simple pocket calculator app.

In the first post, we completed a first draft of the design, using only types (no UML diagrams!). and in the previous post, we created an initial implementation that exercised the design and revealed a missing requirement.

Now it's time to build the remaining components and assemble them into a complete application

Creating the services

We have a implementation. But the implementation depends on some services, and we haven't created the services yet.

In practice though, this bit is very easy and straightforward. The types defined in the domain enforce constraints such there is really only one way of writing the code.

I'm going to show all the code at once (below), and I'll add some comments afterwards.

```
// ================================================
// Implementation of CalculatorConfiguration
// ================================================
module CalculatorConfiguration =

    // A record to store configuration options
    // (e.g. loaded from a file or environment)
    type Configuration = {
        decimalSeparator : string
        divideByZeroMsg : string
        maxDisplayLength: int
        }

    let loadConfig() = {
        decimalSeparator =
            System.Globalization.CultureInfo.CurrentCulture.NumberFormat.CurrencyDecim
alSeparator
        divideByZeroMsg = "ERR-DIV0"
        maxDisplayLength = 10
        }

// ================================================
// Implementation of CalculatorServices
// ================================================
```

```
module CalculatorServices =
    open CalculatorDomain
    open CalculatorConfiguration

    let updateDisplayFromDigit (config:Configuration) :UpdateDisplayFromDigit =
        fun (digit, display) ->

        // determine what character should be appended to the display
        let appendCh=
            match digit with
            | Zero ->
                // only allow one 0 at start of display
                if display="0" then "" else "0"
            | One -> "1"
            | Two -> "2"
            | Three-> "3"
            | Four -> "4"
            | Five -> "5"
            | Six-> "6"
            | Seven-> "7"
            | Eight-> "8"
            | Nine-> "9"
            | DecimalSeparator ->
                if display="" then
                    // handle empty display with special case
                    "0" + config.decimalSeparator
                else if display.Contains(config.decimalSeparator) then
                    // don't allow two decimal separators
                    ""
                else
                    config.decimalSeparator

        // ignore new input if there are too many digits
        if (display.Length > config.maxDisplayLength) then
            display // ignore new input
        else
            // append the new char
            display + appendCh

    let getDisplayNumber :GetDisplayNumber = fun display ->
        match System.Double.TryParse display with
        | true, d -> Some d
        | false, _ -> None

    let setDisplayNumber :SetDisplayNumber = fun f ->
        sprintf "%g" f

    let setDisplayError divideByZeroMsg :SetDisplayError = fun f ->
        match f with
        | DivideByZero -> divideByZeroMsg

    let doMathOperation  :DoMathOperation = fun (op,f1,f2) ->
        match op with
```

```
        | Add -> Success (f1 + f2)
        | Subtract -> Success (f1 - f2)
        | Multiply -> Success (f1 * f2)
        | Divide ->
            try
                Success (f1 / f2)
            with
            | :? System.DivideByZeroException ->
                Failure DivideByZero

    let initState :InitState = fun () ->
        {
        display=""
        pendingOp = None
        }

    let createServices (config:Configuration) = {
        updateDisplayFromDigit = updateDisplayFromDigit config
        doMathOperation = doMathOperation
        getDisplayNumber = getDisplayNumber
        setDisplayNumber = setDisplayNumber
        setDisplayError = setDisplayError (config.divideByZeroMsg)
        initState = initState
        }
```

Some comments:

- I have created a configuration record that stores properties that are used to parameterize the services, such as the decimal separator.
- The configuration record is passed into the `createServices` function, which in turn passes the configuration on those services that need it.
- All the functions use the same approach of returning one of the types defined in the design, such as `UpdateDisplayFromDigit` or `DoMathOperation`.
- There are only a few tricky edge cases, such as trapping exceptions in division, or preventing more than one decimal separator being appended.

Creating the user interface

For the user interface, I'm going to use WinForms rather than WPF or a web-based approach. It's simple and should work on Mono/Xamarin as well as Windows. And it should be easy to port to other UI frameworks as well.

As is typical with UI development I spent more time on this than on any other part of the process! I'm going to spare you all the painful iterations and just go directly to the final version.

I won't show all the code, as it is about 200 lines (and you can see it in the gist), but here are some highlights:

```
module CalculatorUI =

    open CalculatorDomain

    type CalculatorForm(initState:InitState, calculate:Calculate) as this =
        inherit Form()

        // initialization before constructor
        let mutable state = initState()
        let mutable setDisplayedText =
            fun text -> () // do nothing
```

The `CalculatorForm` is a subclass of `Form`, as usual.

There are two parameters for its constructor. One is `initState`, the function that creates an empty state, and `calculate`, the function that transforms the state based on the input. In other words, I'm using standard constructor based dependency injection here.

There are two mutable fields (shock horror!).

One is the state itself. Obviously, it will be modified after each button is pressed.

The second is a function called `setDisplayedText`. What's that all about?

Well, after the state has changed, we need to refresh the control (a Label) that displays the text.

The standard way to do it is to make the label control a field in the form, like this:

```
type CalculatorForm(initState:InitState, calculate:Calculate) as this =
    inherit Form()

    let displayControl :Label = null
```

and then set it to an actual control value when the form has been initialized:

```
member this.CreateDisplayLabel() =
    let display = new Label(Text="",Size=displaySize,Location=getPos(0,0))
    display.TextAlign <- ContentAlignment.MiddleRight
    display.BackColor <- Color.White
    this.Controls.Add(display)

    // traditional style - set the field when the form has been initialized
    displayControl <- display
```

But this has the problem that you might accidentally try to access the label control before it is initialized, causing a NRE. Also, I'd prefer to focus on the desired behavior, rather than having a "global" field that can be accessed by anyone anywhere.

By using a function, we (a) encapsulate the access to the real control and (b) avoid any possibility of a null reference.

The mutable function starts off with a safe default implementation (`fun text -> ()`), and is then changed to a *new* implementation when the label control is created:

```
member this.CreateDisplayLabel() =
    let display = new Label(Text="",Size=displaySize,Location=getPos(0,0))
    this.Controls.Add(display)

    // update the function that sets the text
    setDisplayedText <-
        (fun text -> display.Text <- text)
```

Creating the buttons

The buttons are laid out in a grid, and so I create a helper function `getPos(row,col)` that gets the physical position from a logical (row,col) on the grid.

Here's an example of creating the buttons:

```
member this.CreateButtons() =
    let sevenButton = new Button(Text="7",Size=buttonSize,Location=getPos(1,0),BackCol
or=DigitButtonColor)
    sevenButton |> addDigitButton Seven

    let eightButton = new Button(Text="8",Size=buttonSize,Location=getPos(1,1),BackCol
or=DigitButtonColor)
    eightButton |> addDigitButton Eight

    let nineButton = new Button(Text="9",Size=buttonSize,Location=getPos(1,2),BackColo
r=DigitButtonColor)
    nineButton |> addDigitButton Nine

    let clearButton = new Button(Text="C",Size=buttonSize,Location=getPos(1,3),BackCol
or=DangerButtonColor)
    clearButton |> addActionButton Clear

    let addButton = new Button(Text="+",Size=doubleHeightSize,Location=getPos(1,4),Bac
kColor=OpButtonColor)
    addButton |> addOpButton Add
```

And since all the digit buttons have the same behavior, as do all the math op buttons, I just created some helpers that set the event handler in a generic way:

```
let addDigitButton digit (button:Button) =
    button.Click.AddHandler(EventHandler(fun _ _ -> handleDigit digit))
    this.Controls.Add(button)

let addOpButton op (button:Button) =
    button.Click.AddHandler(EventHandler(fun _ _ -> handleOp op))
    this.Controls.Add(button)
```

I also added some keyboard support:

```
member this.KeyPressHandler(e:KeyPressEventArgs) =
    match e.KeyChar with
    | '0' -> handleDigit Zero
    | '1' -> handleDigit One
    | '2' -> handleDigit Two
    | '.' | ',' -> handleDigit DecimalSeparator
    | '+' -> handleOp Add
    // etc
```

Button clicks and keyboard presses are eventually routed into the key function `handleInput`, which does the calculation.

```
let handleInput input =
     let newState = calculate(input,state)
     state <- newState
     setDisplayedText state.display

let handleDigit digit =
     Digit digit |> handleInput

let handleOp op =
     Op op |> handleInput
```

As you can see, the implementation of `handleInput` is trivial. It calls the calculation function that was injected, sets the mutable state to the result, and then updates the display.

So there you have it -- a complete calculator!

Let's try it now -- get the code from this gist and try running it as a F# script.

Disaster strikes!

Let's start with a simple test. Try entering `1` `Add` `2` `Equals`. What would you expect?

I don't know about you, but what I *wouldn't* expect is that the calculator display shows `12`!

What's going on? Some quick experimenting shows that I have forgotten something really important -- when an `Add` or `Equals` operation happens, any subsequent digits should *not* be added to the current buffer, but instead start a new one. Oh no! We've got a showstopper bug!

Remind me again, what idiot said "if it compiles, it probably works".*

* Actually, that idiot would be me (among many others).

So what went wrong then?

Well the code did compile, but it didn't work as expected, not because the code was buggy, but because *my design was flawed*.

In other words, the use of the types from the type-first design process means that I *do* have high confidence that the code I wrote is a correct implementation of the design. But if the requirements and design are wrong, all the correct code in the world can't fix that.

We'll revisit the requirements in the next post, but meanwhile, is there a patch we can make that will fix the problem?

Fixing the bug

Let's think of the circumstances when we start a new set of digits, vs. when we just append to the existing ones. As we noted above, a math operation or `Equals` will force the reset.

So why not set a flag when those operations happen? If the flag is set, then start a new display buffer, and after that, unset the flag so that characters are appended as before.

What changes do we need to make to the code?

First, we need to store the flag somewhere. We'll store it in the `CalculatorState` of course!

```
type CalculatorState = {
    display: CalculatorDisplay
    pendingOp: (CalculatorMathOp * Number) option
    allowAppend: bool
    }
```

(*This might seem like a good solution for now, but using flags like this is really a design smell. In the next post, I'll use a different approach which doesn't involve flags)*

Fixing the implementation

With this change made, compiling the `CalculatorImplementation` code now breaks everywhere a new state is created.

Actually, that's what I like about using F# -- something like adding a new field to a record is a breaking change, rather than something that can be overlooked by mistake.

We'll make the following tweaks to the code:

- For `updateDisplayFromDigit`, we return a new state with `allowAppend` set to true.
- For `updateDisplayFromPendingOp` and `addPendingMathOp`, we return a new state with `allowAppend` set to false.

Fixing the services

Most of the services are fine. The only service that is broken now is `initState`, which just needs to be tweaked to have `allowAppend` be true when starting.

```
let initState :InitState = fun () ->
    {
    display=""
    pendingOp = None
    allowAppend = true
    }
```

Fixing the user interface

The `CalculatorForm` class continues to work with no changes.

But this change does raise the question of how much the `CalculatorForm` should know about the internals of the `CalculatorDisplay` type.

Should `CalculatorDisplay` be transparent, in which case the form might break every time we change the internals?

Or should `CalculatorDisplay` be an opaque type, in which case we will need to add another "service" that extracts the buffer from the `CalculatorDisplay` type so that the form can display it?

For now, I'm happy to tweak the form if there are changes. But in a bigger or more long-term project, when we are trying to reduce dependencies, then yes, I would make the domain types opaque as much as possible to reduce the fragility of the design.

Testing the patched version

Let's try out the patched version now (*you can get the code for the patched version from this gist*).

Does it work now?

Yes. Entering `1` `Add` `2` `Equals` results in `3`, as expected.

So that fixes the major bug. Phew.

But if you keep playing around with this implementation, you will encounter other ~~bugs~~ undocumented features too.

For example:

- `1.0 / 0.0` displays `Infinity`. What happened to our divide by zero error?
- You get strange behaviors if you enter operations in unusual orders. For example, entering `2 + + -` shows `8` on the display!

So obviously, this code is not yet fit for purpose.

What about Test-Driven Development?

At this point, you might be saying to yourself: "if only he had used TDD this wouldn't have happened".

It's true -- I wrote all this code, and yet I didn't even bother to write a test that checked whether you could add two numbers properly!

If I had started out by writing tests, and letting that drive the design, then surely I wouldn't have run into this problem.

Well in this particular example, yes, I would probably would have caught the problem immediately. In a TDD approach, checking that `1 + 2 = 3` would have been one of the first tests I wrote! But on the other hand, for obvious flaws like this, any interactive testing will reveal the issue too.

To my mind, the advantages of test-driven development are that:

- it drives the *design* of the code, not just the implementation.
- it provides guarantees that code stays correct during refactoring.

So the real question is, would test-driven development help us find missing requirements or subtle edge cases? Not necessarily. Test-driven development will only be effective if we can think of every possible case that could happen in the first place. In that sense, TDD would not make up for a lack of imagination!

And if do have good requirements, then hopefully we can design the types to make illegal states unrepresentable and then we won't need the tests to provide correctness guarantees.

Now I'm not saying that I am against automated testing. In fact, I do use it all the time to verify certain requirements, and especially for integration and testing in the large.

So, for example, here is how I might test this code:

```
module CalculatorTests =
    open CalculatorDomain
    open System

    let config = CalculatorConfiguration.loadConfig()
    let services = CalculatorServices.createServices config
    let calculate = CalculatorImplementation.createCalculate services

    let emptyState = services.initState()

    /// Given a sequence of inputs, start with the empty state
    /// and apply each input in turn. The final state is returned
    let processInputs inputs =
        // helper for fold
        let folder state input =
            calculate(input,state)

        inputs
        |> List.fold folder emptyState

    /// Check that the state contains the expected display value
    let assertResult testLabel expected state =
        let actual = state.display
        if (expected <> actual) then
            printfn "Test %s failed: expected=%s actual=%s" testLabel expected actual
        else
            printfn "Test %s passed" testLabel

    let ``when I input 1 + 2, I expect 3``() =
        [Digit One; Op Add; Digit Two; Action Equals]
        |> processInputs
        |> assertResult "1+2=3" "3"

    let ``when I input 1 + 2 + 3, I expect 6``() =
        [Digit One; Op Add; Digit Two; Op Add; Digit Three; Action Equals]
        |> processInputs
        |> assertResult "1+2+3=6" "6"

    // run tests
    do
        ``when I input 1 + 2, I expect 3``()
        ``when I input 1 + 2 + 3, I expect 6``()
```

And of course, this would be easily adapted to using NUnit or similar.

How can I develop a better design?

I messed up! As I said earlier, the *implementation itself* was not the problem. I think the type-first design process worked. The real problem was that I was too hasty and just dived into the design without really understanding the requirements.

How can I prevent this from happening again next time?

One obvious solution would be to switch to a proper TDD approach. But I'm going to be a bit stubborn, and see if I can stay with a type-first design!

In the next post, I will stop being so ad-hoc and over-confident, and instead use a process that is more thorough and much more likely to prevent these kinds of errors at the design stage.

The code for this post is available on GitHub in this gist (unpatched) and this gist (patched).

Calculator Walkthrough: Part 4

In this series of posts, I've been developing a simple pocket calculator app.

In the first post, we completed a first draft of the design, using type-first development. and in the second post, we created an initial implemeentation.

In the previous post, we created the rest of the code, including the user interface, and attempted to use it.

But the final result was unusable! The problem wasn't that the code was buggy, it was that I didn't spend enough time thinking about the requirements before I started coding!

Oh well. As Fred Brooks famously said: "plan to throw one away; you will, anyhow" (although that is a bit simplistic).

The good news is that I have learned from the previous bad implementation, and have a plan to make the design better.

Reviewing the bad design

Looking at the design and implementation (see this gist), a few things stand out:

First, the event handling types such as `UpdateDisplayFromDigit` did not take into account the *context*, the current state of the calculator. The `allowAppend` flag we added as a patch was one way to take the context into account, but it smells awful bad.

Second there was a bit of special case code for certain inputs (`Zero` and `DecimalSeparator`), as you can see from this code snippet:

```
let appendCh=
    match digit with
    | Zero ->
        // only allow one 0 at start of display
        if display="0" then "" else "0"
    | One -> "1"
    | // snip
    | DecimalSeparator ->
        if display="" then
            // handle empty display with special case
            "0" + config.decimalSeparator
        else if display.Contains(config.decimalSeparator) then
            // don't allow two decimal separators
            ""
        else
            config.decimalSeparator
```

This makes me think that these inputs should be treated as different *in the design itself* and not hidden in the implementation -- after all we want the design to also act as documentation as much as possible.

Using a finite state machine as a design tool

So if the ad-hoc, make-it-up-as-you-go-along approach failed, what should I do instead?

Well, I am a big proponent of using finite state machines ("FSMs" -- not be confused with the True FSM) where appropriate. It is amazing how often a program can be modelled as a state machine.

What are the benefits of using state machines? I'm going to repeat what I said in another post.

Each state can have different allowable behavior. In other words, a state machine forces you to think about context, and what options are available in that context.

In this case, I forgot that the context changed after an `Add` was processed, and thus the rules for accumulating digits changed too.

All the states are explicitly documented. It is all too easy to have important states that are implicit but never documented.

For example, I have created special code to deal with zero and decimal separators. Currently it is buried away in the implementation, but it should be part of the design.

It is a design tool that forces you to think about every possibility that could occur. A common cause of errors is that certain edge cases are not handled, but a state machine forces *all* cases to be thought about.

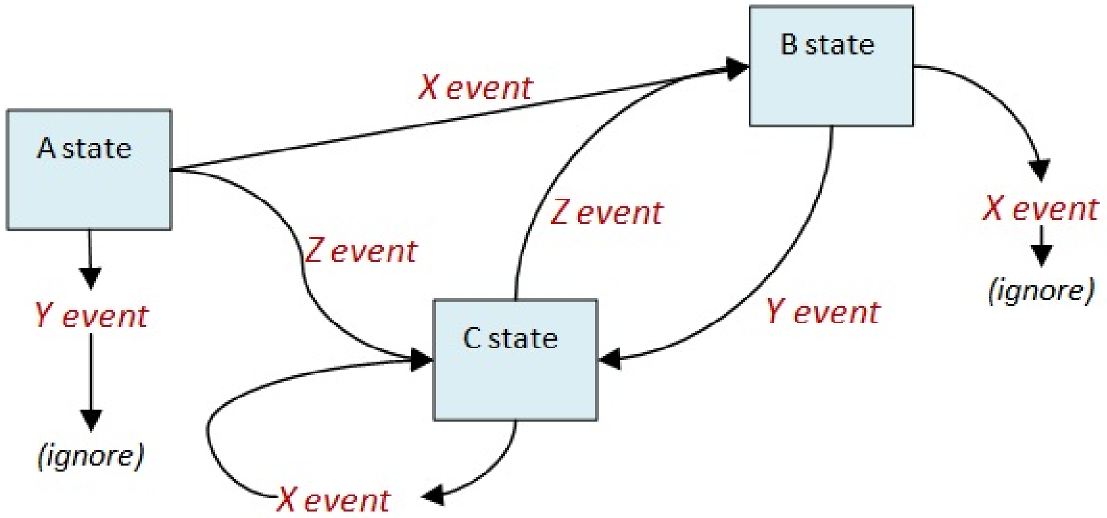
In this case, in addition to the most obvious bug, there are still some edge cases that are not dealt with properly, such as immediately following a math operation with *another* math operation. What should happen then?

How to implement simple finite state machines in F#

You are probably familiar with complex FSMs, such as those used in language parsers and regular expressions. Those kinds of state machines are generated from rule sets or grammars, and are quite complicated.

The kinds of state machines that I'm talking about are much, much simpler. Just a few cases at the most, with a small number of transitions, so we don't need to use complex generators.

Here's an example of what I am talking about:



So what is the best way to implement these simple state machines in F#?

Now, designing and implementing FSMs is a complex topic in in own right, with its own terminology (NFAs and DFAs, Moore vs. Mealy, etc), and whole businesses built around it.

In F#, there are a number of possible approaches, such as table driven, or mutually recursive functions, or agents, or OO-style subclasses, etc.

But my preferred approach (for an ad-hoc manual implementation) makes extensive use of union types and pattern matching.

First, create a union type that represents all the states. For example, if there are three states called "A", "B" and "C", the type would look like this:

```
type State =
    | AState
    | BState
    | CState
```

In many cases, each state will need to store some data that is relevant to that state. So we will need to create types to hold that data as well.

```
type State =
    | AState of AStateData
    | BState of BStateData
    | CState
and AStateData =
    {something:int}
and BStateData =
    {somethingElse:int}
```

Next, all possible events that can happen are defined in another union type. If events have data associated with them, add that as well.

```
type InputEvent =
    | XEvent
    | YEvent of YEventData
    | ZEvent
and YEventData =
    {eventData:string}
```

Finally, we can create a "transition" function that, given a current state and input event, returns a new state.

```
let transition (currentState,inputEvent) =
    match currentState,inputEvent with
    | AState, XEvent -> // new state
    | AState, YEvent -> // new state
    | AState, ZEvent -> // new state
    | BState, XEvent -> // new state
    | BState, YEvent -> // new state
    | CState, XEvent -> // new state
    | CState, ZEvent -> // new state
```

What I like about this approach in a language with pattern matching, like F#, is that **if we forget to handle a particular combination of state and event, we get a compiler warning**. How awesome is that?

It's true that, for systems with many states and input events, it may be unreasonable to expect every possible combination to be explicitly handled. But in my experience, many nasty bugs are caused by processing an event when you shouldn't, exactly as we saw with the original design accumulating digits when it shouldn't have.

Forcing yourself to consider every possible combination is thus a helpful design practice.

Now, even with a small number of states and events, the number of possible combinations gets large very quickly. To make it more manageable in practice, I typically create a series of helper functions, one for each state, like this:

```
let aStateHandler stateData inputEvent =
    match inputEvent with
    | XEvent -> // new state
    | YEvent _ -> // new state
    | ZEvent -> // new state

let bStateHandler stateData inputEvent =
    match inputEvent with
    | XEvent -> // new state
    | YEvent _ -> // new state
    | ZEvent -> // new state

let cStateHandler inputEvent =
    match inputEvent with
    | XEvent -> // new state
    | YEvent _ -> // new state
    | ZEvent -> // new state

let transition (currentState,inputEvent) =
    match currentState with
    | AState stateData ->
        // new state
        aStateHandler stateData inputEvent
    | BState stateData ->
        // new state
        bStateHandler stateData inputEvent
    | CState ->
        // new state
        cStateHandler inputEvent
```

So let's try this approach and attempt to implement the state diagram above:

```
let aStateHandler stateData inputEvent =
    match inputEvent with
    | XEvent ->
        // transition to B state
        BState {somethingElse=stateData.something}
    | YEvent _ ->
        // stay in A state
        AState stateData
    | ZEvent ->
        // transition to C state
        CState

let bStateHandler stateData inputEvent =
    match inputEvent with
    | XEvent ->
        // stay in B state
        BState stateData
    | YEvent _ ->
        // transition to C state
        CState

let cStateHandler inputEvent =
    match inputEvent with
    | XEvent ->
        // stay in C state
        CState
    | ZEvent ->
        // transition to B state
        BState {somethingElse=42}

let transition (currentState,inputEvent) =
    match currentState with
    | AState stateData ->
        aStateHandler stateData inputEvent
    | BState stateData ->
        bStateHandler stateData inputEvent
    | CState ->
        cStateHandler inputEvent
```

If we try to compile this, we immediately get some warnings:

- (near bStateHandler) `Incomplete pattern matches on this expression. For example, the value 'ZEvent' may indicate a case not covered by the pattern(s).`
- (near cStateHandler) `Incomplete pattern matches on this expression. For example, the value 'YEvent (_)' may indicate a case not covered by the pattern(s).`

This is really helpful. It means we have missed some edge cases and we should change our code to handle these events.

By the way, please do *not* fix the code with a wildcard match (underscore)! That defeats the purpose. If you want to ignore an event, do it explicitly.

Here's the fixed up code, which compiles without warnings:

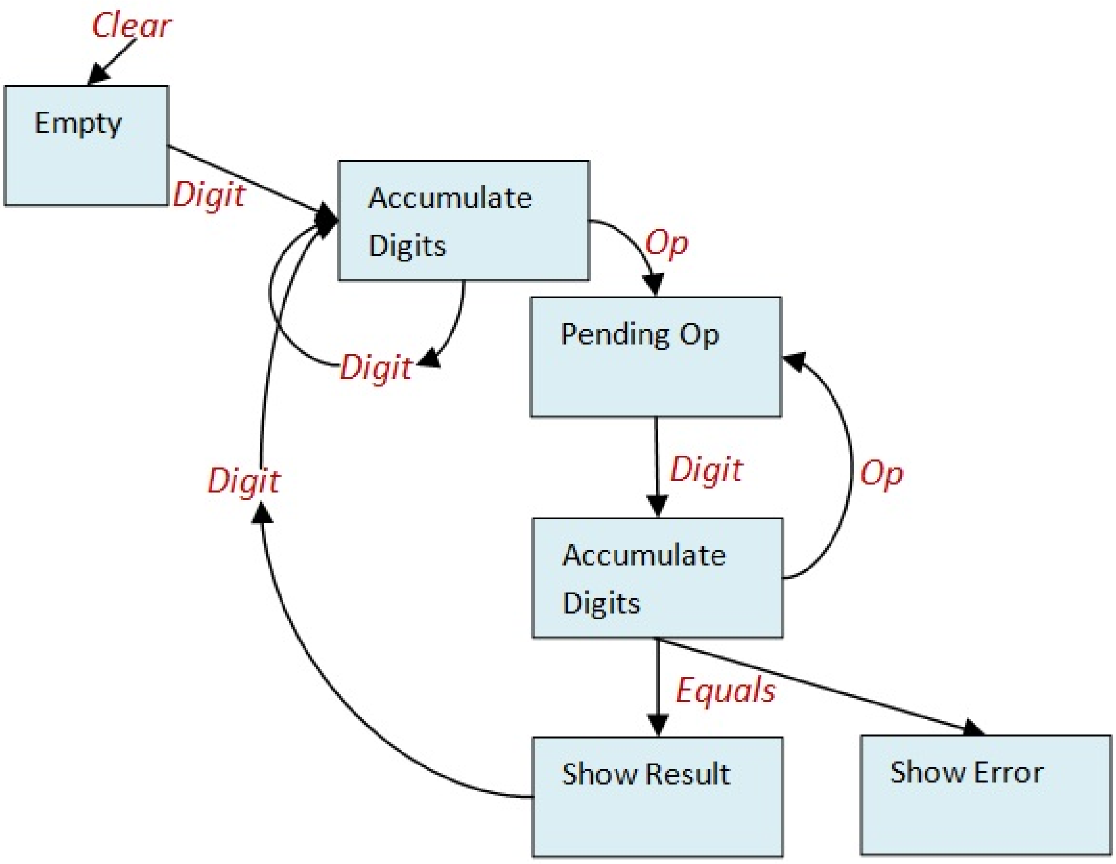
```
let bStateHandler stateData inputEvent =
    match inputEvent with
    | XEvent
    | ZEvent ->
        // stay in B state
        BState stateData
    | YEvent _ ->
        // transition to C state
        CState

let cStateHandler inputEvent =
    match inputEvent with
    | XEvent
    | YEvent _ ->
        // stay in C state
        CState
    | ZEvent ->
        // transition to B state
        BState {somethingElse=42}
```

You can see the code for this example in this gist.

Designing the state machine for the calculator

Let's sketch out a state machine for the calculator now. Here's a first attempt:



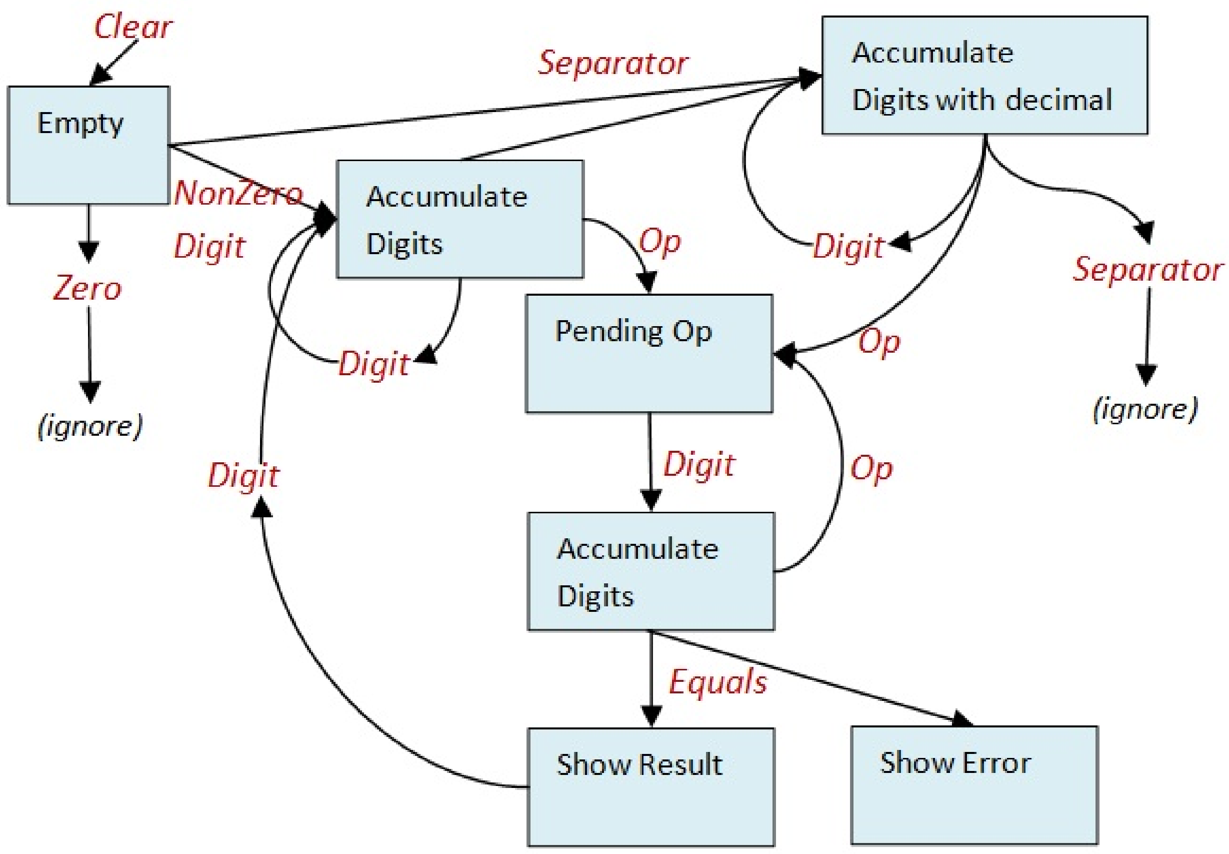
Each state is a box, and the events that trigger transitions (such as a digit or math operation or `Equals`) are in red.

If we follow through a sequence of events for something like `1` `Add` `2` `Equals`, you can see that we'll end up at the "Show result" state at the bottom.

But remember that we wanted to raise the handling of zero and decimal separators up to the design level?

So let's create special events for those inputs, and a new state "accumulate with decimal" that ignores subsequent decimal separators.

Here's version 2:



Finalizing the state machine

> "Good artists copy. Great artists steal." -- Pablo Picasso (but not really)

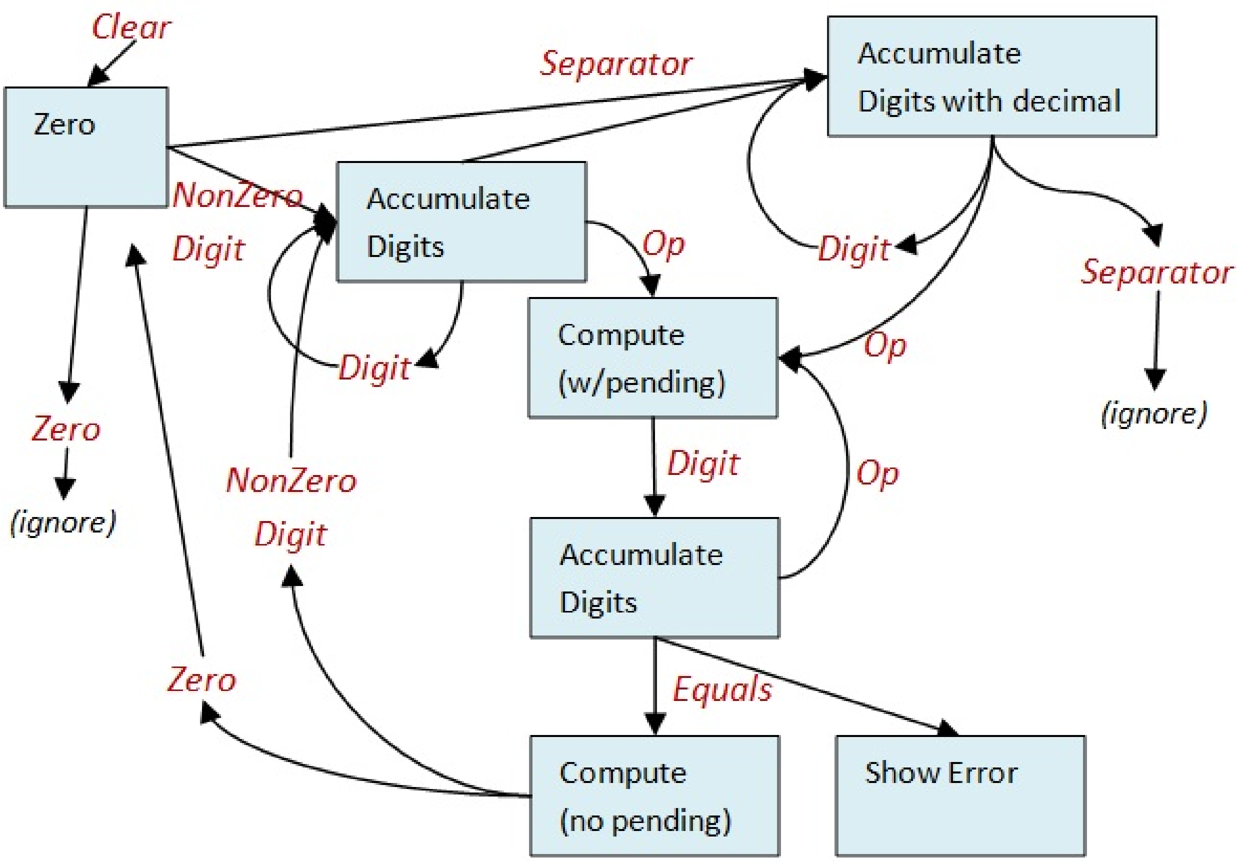
At this point, I'm thinking that surely I can't be only person to have thought of using a state machine to model a calculator? Perhaps I can do some research and ~~steal~~ borrow someone else's design?

Sure enough, googling for "calculator state machine" brings up all sorts of results, including this one which has a detailed spec and state transition diagram.

Looking at that diagram, and doing some more thinking, leads to the following insights:

- The "clear" state and zero state are the same. Sometimes there is a pending op, sometimes not.
- A math operation and `Equals` are very similar in that they update the display with any pending calculation. The only difference is whether a pending op is added to the state or not.
- The error message case definitely needs to be a distinct state. It ignores all input other than `Clear`.

With these insights in mind then, here's version 3 of our state transition diagram:



I'm only showing the key transitions -- it would be too overwhelming to show all of them. But it does give us enough information to get started on the detailed requirements.

As we can see, there are five states:

- ZeroState
- AccumulatorState
- AccumulatorDecimalState
- ComputedState
- ErrorState

And there are six possible inputs:

- Zero
- NonZeroDigit
- DecimalSeparator
- MathOp
- Equals
- Clear

Let's document each state, and what data it needs to store, if any.

State	Data associated with state	Special behavior?
ZeroState	(optional) pending op	Ignores all Zero input
AccumulatorState	buffer and (optional) pending op	Accumulates digits in buffer
AccumulatorDecimalState	buffer and (optional) pending op	Accumulates digits in buffer, but ignores decimal separators
ComputedState	Calculated number and (optional) pending op	
ErrorState	Error message	Ignores all input other than Clear

Documenting each state and event combination

Next we should think about what happens for each state and event combination. As with the sample code above, we'll group them so that we only have to deal with the events for one state at a time.

Let's start with the `ZeroState` state. Here are the transitions for each type of input:

Input	Action	New State
Zero	(ignore)	ZeroState
NonZeroDigit	Start a new accumulator with the digit.	AccumulatorState
DecimalSeparator	Start a new accumulator with "0."	AccumulatorDecimalState
MathOp	Go to Computed or ErrorState state. If there is a pending op, update the display based on the result of the calculation (or error). Also, if calculation was successful, push a new pending op, built from the event, using a current number of "0".	ComputedState
Equals	As with MathOp, but without any pending op	ComputedState
Clear	(ignore)	ZeroState

We can repeat the process with the `AccumulatorState` state. Here are the transitions for each type of input:

Input	Action	New State
Zero	Append "0" to the buffer.	AccumulatorState
NonZeroDigit	Append the digit to the buffer.	AccumulatorState
DecimalSeparator	Append the separator to the buffer, and transition to new state.	AccumulatorDecimalState
MathOp	Go to Computed or ErrorState state. If there is a pending op, update the display based on the result of the calculation (or error). Also, if calculation was successful, push a new pending op, built from the event, using a current number based on whatever is in the accumulator.	ComputedState
Equals	As with MathOp, but without any pending op	ComputedState
Clear	Go to Zero state. Clear any pending op.	ZeroState

The event handling for `AccumulatorDecimalState` state is the same, except that `DecimalSeparator` is ignored.

What about the `ComputedState` state. Here are the transitions for each type of input:

Input	Action	New State
Zero	Go to ZeroState state, but preserve any pending op	ZeroState
NonZeroDigit	Start a new accumulator, preserving any pending op	AccumulatorState
DecimalSeparator	Start a new decimal accumulator, preserving any pending op	AccumulatorDecimalState
MathOp	Stay in Computed state. Replace any pending op with a new one built from the input event	ComputedState
Equals	Stay in Computed state. Clear any pending op	ComputedState
Clear	Go to Zero state. Clear any pending op.	ZeroState

Finally, the `ErrorState` state is very easy. :

Input	Action	New State
Zero, NonZeroDigit, DecimalSeparator MathOp, Equals	(ignore)	ErrorState
Clear	Go to Zero state. Clear any pending op.	ZeroState

Converting the states into F# code

Now that we've done all this work, the conversion into types is straightforward.

Here are the main types:

```
type Calculate = CalculatorInput * CalculatorState -> CalculatorState
// five states
and CalculatorState =
    | ZeroState of ZeroStateData
    | AccumulatorState of AccumulatorStateData
    | AccumulatorWithDecimalState of AccumulatorStateData
    | ComputedState of ComputedStateData
    | ErrorState of ErrorStateData
// six inputs
and CalculatorInput =
    | Zero
    | Digit of NonZeroDigit
    | DecimalSeparator
    | MathOp of CalculatorMathOp
    | Equals
    | Clear
// data associated with each state
and ZeroStateData =
    PendingOp option
and AccumulatorStateData =
    {digits:DigitAccumulator; pendingOp:PendingOp option}
and ComputedStateData =
    {displayNumber:Number; pendingOp:PendingOp option}
and ErrorStateData =
    MathOperationError
```

If we compare these types to the first design (below), we have now made it clear that there is something special about `Zero` and `DecimalSeparator`, as they have been promoted to first class citizens of the input type.

```
// from the old design
type CalculatorInput =
    | Digit of CalculatorDigit
    | Op of CalculatorMathOp
    | Action of CalculatorAction

// from the new design
type CalculatorInput =
    | Zero
    | Digit of NonZeroDigit
    | DecimalSeparator
    | MathOp of CalculatorMathOp
    | Equals
    | Clear
```

Also, in the old design, we had a single state type (below) that stored data for all contexts, while in the new design, the state is *explicitly different* for each context. The types `ZeroStateData` , `AccumulatorStateData` , `ComputedStateData` , and `ErrorStateData` make this obvious.

```
// from the old design
type CalculatorState = {
    display: CalculatorDisplay
    pendingOp: (CalculatorMathOp * Number) option
    }

// from the new design
type CalculatorState =
    | ZeroState of ZeroStateData
    | AccumulatorState of AccumulatorStateData
    | AccumulatorWithDecimalState of AccumulatorStateData
    | ComputedState of ComputedStateData
    | ErrorState of ErrorStateData
```

Now that we have the basics of the new design, we need to define the other types referenced by it:

```
and DigitAccumulator = string
and PendingOp = (CalculatorMathOp * Number)
and Number = float
and NonZeroDigit=
    | One | Two | Three | Four
    | Five | Six | Seven | Eight | Nine
and CalculatorMathOp =
    | Add | Subtract | Multiply | Divide
and MathOperationResult =
    | Success of Number
    | Failure of MathOperationError
and MathOperationError =
    | DivideByZero
```

And finally, we can define the services:

```
// services used by the calculator itself
type AccumulateNonZeroDigit = NonZeroDigit * DigitAccumulator -> DigitAccumulator
type AccumulateZero = DigitAccumulator -> DigitAccumulator
type AccumulateSeparator = DigitAccumulator -> DigitAccumulator
type DoMathOperation = CalculatorMathOp * Number * Number -> MathOperationResult
type GetNumberFromAccumulator = AccumulatorStateData -> Number

// services used by the UI or testing
type GetDisplayFromState = CalculatorState -> string
type GetPendingOpFromState = CalculatorState -> string

type CalculatorServices = {
    accumulateNonZeroDigit :AccumulateNonZeroDigit
    accumulateZero :AccumulateZero
    accumulateSeparator :AccumulateSeparator
    doMathOperation :DoMathOperation
    getNumberFromAccumulator :GetNumberFromAccumulator
    getDisplayFromState :GetDisplayFromState
    getPendingOpFromState :GetPendingOpFromState
    }
```

Note that because the state is much more complicated, I've added helper function `getDisplayFromState` that extracts the display text from the state. This helper function will be used the UI or other clients (such as tests) that need to get the text to display.

I've also added a `getPendingOpFromState`, so that we can show the pending state in the UI as well.

Creating a state-based implementation

Now we can create a state-based implementation, using the pattern described earlier.

(The complete code is available in this gist.)

Let's start with the main function that does the state transitions:

```
let createCalculate (services:CalculatorServices) :Calculate =
    // create some local functions with partially applied services
    let handleZeroState = handleZeroState services
    let handleAccumulator = handleAccumulatorState services
    let handleAccumulatorWithDecimal = handleAccumulatorWithDecimalState services
    let handleComputed = handleComputedState services
    let handleError = handleErrorState

    fun (input,state) ->
        match state with
        | ZeroState stateData ->
            handleZeroState stateData input
        | AccumulatorState stateData ->
            handleAccumulator stateData input
        | AccumulatorWithDecimalState stateData ->
            handleAccumulatorWithDecimal stateData input
        | ComputedState stateData ->
            handleComputed stateData input
        | ErrorState stateData ->
            handleError stateData input
```

As you can see, it passes the responsibility to a number of handlers, one for each state, which will be discussed below.

But before we do that, I thought it might be instructive to compare the new state-machine based design with the (buggy!) one I did previously.

Here is the code from the previous one:

```
let createCalculate (services:CalculatorServices) :Calculate =
    fun (input,state) ->
        match input with
        | Digit d ->
            let newState = updateDisplayFromDigit services d state
            newState //return
        | Op op ->
            let newState1 = updateDisplayFromPendingOp services state
            let newState2 = addPendingMathOp services op newState1
            newState2 //return
        | Action Clear ->
            let newState = services.initState()
            newState //return
        | Action Equals ->
            let newState = updateDisplayFromPendingOp services state
            newState //return
```

If we compare the two implementations, we can see that there has been a shift of emphasis from events to state. You can see this by comparing how main pattern matching is done in the two implementations:

- In the original version, the focus was on the input, and the state was secondary.
- In the new version, the focus is on the state, and the input is secondary.

The focus on *input* over *state*, ignoring the context, is why the old version was such a bad design.

To repeat what I said above, many nasty bugs are caused by processing an event when you shouldn't (as we saw with the original design). I feel much more confident in the new design because of the explicit emphasis on state and context from the very beginning.

In fact, I'm not alone in noticing these kinds of issues. Many people think that classic "event-driven programming" is flawed and recommend a more "state driven approach" (e.g. here and here), just as I have done here.

Creating the handlers

We have already documented the requirements for each state transition, so writing the code is straightforward. We'll start with the code for the `ZeroState` handler:

```
let handleZeroState services pendingOp input =
    // create a new accumulatorStateData object that is used when transitioning to oth
er states
    let accumulatorStateData = {digits=""; pendingOp=pendingOp}
    match input with
    | Zero ->
        ZeroState pendingOp // stay in ZeroState
    | Digit digit ->
        accumulatorStateData
        |> accumulateNonZeroDigit services digit
        |> AccumulatorState  // transition to AccumulatorState
    | DecimalSeparator ->
        accumulatorStateData
        |> accumulateSeparator services
        |> AccumulatorWithDecimalState  // transition to AccumulatorWithDecimalState
    | MathOp op ->
        let nextOp = Some op
        let newState = getComputationState services accumulatorStateData nextOp
        newState  // transition to ComputedState or ErrorState
    | Equals ->
        let nextOp = None
        let newState = getComputationState services accumulatorStateData nextOp
        newState  // transition to ComputedState or ErrorState
    | Clear ->
        ZeroState None // transition to ZeroState and throw away any pending ops
```

Again, the *real* work is done in helper functions such as `accumulateNonZeroDigit` and `getComputationState`. We'll look at those in a minute.

Here is the code for the `AccumulatorState` handler:

```
let handleAccumulatorState services stateData input =
    match input with
    | Zero ->
        stateData
        |> accumulateZero services
        |> AccumulatorState  // stay in AccumulatorState
    | Digit digit ->
        stateData
        |> accumulateNonZeroDigit services digit
        |> AccumulatorState  // stay in AccumulatorState
    | DecimalSeparator ->
        stateData
        |> accumulateSeparator services
        |> AccumulatorWithDecimalState  // transition to AccumulatorWithDecimalState
    | MathOp op ->
        let nextOp = Some op
        let newState = getComputationState services stateData nextOp
        newState  // transition to ComputedState or ErrorState
    | Equals ->
        let nextOp = None
        let newState = getComputationState services stateData nextOp
        newState  // transition to ComputedState or ErrorState
    | Clear ->
        ZeroState None // transition to ZeroState and throw away any pending op
```

Here is the code for the `ComputedState` handler:

```
let handleComputedState services stateData input =
    let emptyAccumulatorStateData = {digits=""; pendingOp=stateData.pendingOp}
    match input with
    | Zero ->
        ZeroState stateData.pendingOp  // transition to ZeroState with any pending op
    | Digit digit ->
        emptyAccumulatorStateData
        |> accumulateNonZeroDigit services digit
        |> AccumulatorState  // transition to AccumulatorState
    | DecimalSeparator ->
        emptyAccumulatorStateData
        |> accumulateSeparator services
        |> AccumulatorWithDecimalState  // transition to AccumulatorWithDecimalState
    | MathOp op ->
        // replace the pending op, if any
        let nextOp = Some op
        replacePendingOp stateData nextOp
    | Equals ->
        // replace the pending op, if any
        let nextOp = None
        replacePendingOp stateData nextOp
    | Clear ->
        ZeroState None // transition to ZeroState and throw away any pending op
```

The helper functions

Finally, let's look at the helper functions:

The accumulator helpers are trivial -- they just call the appropriate service and wrap the result in an `AccumulatorData` record.

```
let accumulateNonZeroDigit services digit accumulatorData =
    let digits = accumulatorData.digits
    let newDigits = services.accumulateNonZeroDigit (digit,digits)
    let newAccumulatorData = {accumulatorData with digits=newDigits}
    newAccumulatorData // return
```

The `getComputationState` helper is much more complex -- the most complex function in the entire code base, I should think.

It's very similar to the `updateDisplayFromPendingOp` that we implemented before, but there are a couple of changes:

- The `services.getNumberFromAccumulator` code can never fail, because of the state-based approach. That makes life simpler!
- The `match result with Success/Failure` code now returns *two* possible states: `ComputedState` or `ErrorState`.
- If there is no pending op, we *still* need to return a valid `ComputedState`, which is what `computeStateWithNoPendingOp` does.

```
let getComputationState services accumulatorStateData nextOp =

    // helper to create a new ComputedState from a given displayNumber
    // and the nextOp parameter
    let getNewState displayNumber =
        let newPendingOp =
            nextOp |> Option.map (fun op -> op,displayNumber )
        {displayNumber=displayNumber; pendingOp = newPendingOp }
        |> ComputedState

    let currentNumber =
        services.getNumberFromAccumulator accumulatorStateData

    // If there is no pending op, create a new ComputedState using the currentNumber
    let computeStateWithNoPendingOp =
        getNewState currentNumber

    maybe {
        let! (op,previousNumber) = accumulatorStateData.pendingOp
        let result = services.doMathOperation(op,previousNumber,currentNumber)
        let newState =
            match result with
            | Success resultNumber ->
                // If there was a pending op, create a new ComputedState using the res
ult
                getNewState resultNumber
            | Failure error ->
                error |> ErrorState
        return newState
        } |> ifNone computeStateWithNoPendingOp
```

Finally, we have a new piece of code that wasn't in the previous implementation at all!

What do you do when you get two math ops in a row? We just replace the old pending op (if any) with the new one (if any).

```
let replacePendingOp (computedStateData:ComputedStateData) nextOp =
    let newPending = maybe {
        let! existing,displayNumber  = computedStateData.pendingOp
        let! next = nextOp
        return next,displayNumber
        }
    {computedStateData with pendingOp=newPending}
    |> ComputedState
```

Completing the calculator

To complete the application, we just need to implement the services and the UI, in the same way as we did before.

As it happens, we can reuse almost all of the previous code. The only thing that has really changed is the way that the input events are structured, which affects how the button handlers are created.

You can get the code for the state machine version of the calculator here.

If you try it out the new code, I think that you will find that it works first time, and feels much more robust. Another win for state-machine driven design!

Exercises

If you liked this design, and want to work on something similar, here are some exercises that you could do:

- First, you could add some other operations. What would you have to change to implement unary ops such as `1/x` and `sqrt`?
- Some calculators have a back button. What would you have to do to implement this? Luckily all the data structures are immutable, so it should be easy!
- Most calculators have a one-slot memory with store and recall. What would you have to change to implement this?
- The logic that says that there are only 10 chars allowed on the display is still hidden from the design. How would you make this visible?

Summary

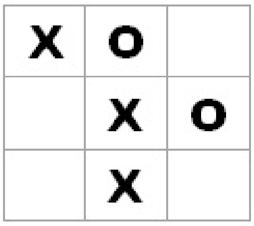
I hope you found this little experiment useful. I certainly learned something, namely: don't shortcut requirements gathering, and consider using a state based approach from the beginning -- it might save you time in the long run!

Enterprise Tic-Tac-Toe

UPDATE: Slides and video from my talk on this topic

This post is one of series in I which I hope to close the gap between theory and practice in functional programming. I pick a small project and show you my thought processes as I go about designing and implementing it from beginning to end.

For the next project in this series of posts, I'm going to do a walkthrough of a Tic-Tac-Toe (aka Noughts and Crosses) implementation, written in a functional style.



Now, to be clear, I'm not a games developer in any shape or form, so I won't be focused on performance or UX at all, just on the design process -- taking some requirements that we all know (I hope) and translating them to functional code.

In fact, to be very clear, I'll deliberately be going a bit overboard on the design just to demonstrate what you can do. There will be no objects. Everything will be immutable, Everything will be typed. There will be capability based security, and more. Performance will *definitely* be taking a back seat. Luckily, Tic-Tac-Toe does not need to support a high frame rate!

In fact, I'm going to call this version "Enterprise Tic-Tac-Toe"!

Why? Well let's look at what you need for "Enterprise":

- We need **separation of concerns** so that specialist teams can work on different parts of the code at the same time.
- We need **a documented API** so that the different teams can work effectively in parallel.
- We need a **security model** to prevent unauthorized actions from occurring.
- We need **well-documented code** so that the architect can ensure that the implementation matches the UML diagrams.
- We need **auditing and logging** to ensure that the system is SOX compliant.
- We need **scalability** to ensure that the system is ready for the challenges of rapid customer acquisition.

Actually, those are the *stated* reasons, but we all know that this is not the whole story. The *real* reasons for an "enterprise design" become apparent when you talk to the people involved:

- *Development Manager:* "We need separation of concerns because the front-end team and back-end team hate each other and refuse to work in the same room."
- *Front-end team:* "We need a documented API so that those dummies building the back-end won't keep breaking our code on every commit."
- *Back-end team:* "We need a security model because those idiots building the front-end will always find a way to do something stupid unless we constrain them."
- *Maintenance team:* "We need well-documented code because we're fed up of having to reverse engineer the hacked-up spaghetti being thrown at us."
- *Testers and Operations:* "We need auditing and logging so that we can see what the effing system is doing inside."
- *Everyone:* "We don't really need scalability at all, but the CTO wants to us to be buzzword compliant."

It's true that there are already some wonderful "enterprise" projects out there, such as Easy Plus in PHP and Fizz Buzz Enterprise Edition in Java, but I hope that my own small contribution to this genre will be considered worthy.

Seriously, I hope that the code won't be quite as ~~bad~~ amusing as those other enterprise projects. In fact, I hope to demonstrate that you can have "enterprise" ready functional code which is still readable!

Designing the domain

> "Write the game such that someone unfamiliar with it could learn the rules by looking at the source code" -- Raganwald

As always, let's use a type-first design. If you recall, this approach means that:

- We start with types only -- no implementation code.
- Every use-case or scenario corresponds to a function type, with one input and one output (which means I'll use tuples when multiple parameters are needed).
- We work mostly top-down and outside-in, but occasionally bottom up as well.
- We ignore the UI for now. And there will be no events or observables in the design of the core domain. It will be purely functional.

In fact, an alternative title for this post might be **growing functional software, guided by types.**

As I have said before, I like to drive the design by working from the events that can happen, rather than the objects involved. I'm old school, so I call them use-cases, but I also like the event-storming approach.

Either way, for the Tic-Tac-Toe "domain", we have three different "event-driven use-cases" (in this case, just various mouse clicks!) to think about:

- Initialize a game
- Player X moves
- Player O moves

Let's start with the first: initialization. This is equivalent to a `new`-style constructor in an OO program.

For Tic-Tac-Toe, there are no configuration parameters needed, so the input would be "null" (aka `unit`) and the output would be a game ready to play, like this:

```
type InitGame = unit -> Game
```

Now, what is this `Game`? Since everything is immutable, the other scenarios are going to have to take an existing game as input, and return a slightly changed version of the game. So `Game` is not quite appropriate. How about `GameState` instead? A "player X moves" function will thus look something like this:

```
type PlayerXMoves = GameState * SomeOtherStuff -> GameState
```

You'll see that I added `SomeOtherStuff` to the input parameters because there's *always* some other stuff! We'll worry about what the "other stuff" is later.

Ok, What should we do next? Should we look more deeply into the internals of `GameState`?

No. Let's stay high-level and do more "outside-in" style design. I like this approach in general because it allows me to focus on what's important and not get side-tracked by implementation details.

Designing the move functions

I said originally that we should have a function for each scenario. Which means we would have functions like this:

```
type PlayerXMoves = GameState * SomeOtherStuff -> GameState
type PlayerOMoves = GameState * SomeOtherStuff -> GameState
```

For each player's move, we start with the current game state, plus some other input created by the player, and end up with a *new* game state.

The problem is that both functions look exactly the same and could be easily substituted for each other. To be honest, I don't trust the user interface to always call the right one -- or at least, it could be a potential issue.

One approach is to have only *one* function, rather than *two*. That way there's nothing to go wrong.

But now we need to handle the two different input cases. How to do that? Easy! A discriminated union type:

```
type UserAction =
    | PlayerXMoves of SomeStuff
    | PlayerOMoves of SomeStuff
```

And now, to process a move, we just pass the user action along with the state, like this:

```
type Move = UserAction * GameState -> GameState
```

So now there is only *one* function for the UI to call rather than two, and less to get wrong.

This approach is great where there is one user, because it documents all the things that they can do. For example, in other games, you might have a type like this:

```
type UserAction =
    | MoveLeft
    | MoveRight
    | Jump
    | Fire
```

However in this situation, this way doesn't feel quite right. Since there are *two* players, what I want to do is give each player their own distinct function to call and not allow them to use the other player's function. This not only stops the user interface component from messing up, but also gives me my capability-based security!

But now we are back to the original problem: how can we tell the two functions apart?

What I'll do is to use types to distinguish them. We'll make the `SomeOtherStuff` be *owned* by each player, like this:

```
type PlayerXMoves = GameState * PlayerX's Stuff -> GameState
type PlayerOMoves = GameState * PlayerO's Stuff -> GameState
```

This way the two functions are distinct, and also PlayerO cannot call PlayerX's function without having some of PlayerX's `Stuff` as well. If this sound's complicated, stay tuned -- it's easier than it looks!

What is SomeOtherStuff?

What is this mysterious `SomeOtherStuff`? In other words, what information do we need to make a move?

For most domains, there might quite a lot of stuff that needs to be passed in, and the stuff might vary based on the context and the state of the system.

But for Tic-Tac-Toe, it's easy, it's just the location on the grid where the player makes their mark. "Top Left", "Bottom Center", and so on.

How should we define this position using a type?

The most obvious approach would be to use a 2-dimensional grid indexed by integers: `(1,1) (1,2) (1,3)`, etc. But I have to admit that I'm too lazy to write unit tests that deal with bounds-checking, nor can I ever remember which integer in the pair is the row and which the column. I want to write code that I don't have to test!

Instead, let's define a type explicitly listing each position of horizontally and vertically:

```
type HorizPosition = Left | HCenter | Right
type VertPosition = Top | VCenter | Bottom
```

And then the position of a square in the grid (which I'm going to call a "cell") is just a pair of these:

```
type CellPosition = HorizPosition * VertPosition
```

If we go back to the "move function" definitions, we now have:

```
type PlayerXMoves = GameState * CellPosition -> GameState
type PlayerOMoves = GameState * CellPosition -> GameState
```

which means: "to play a move, the input is a game state and a selected cell position, and the output is an updated game state".

Both player X and player O can play the *same* cell position, so, as we said earlier, we need to make them distinct.

I'm going to do that by wrapping them in a single case union:

```
type PlayerXPos = PlayerXPos of CellPosition
type PlayerOPos = PlayerOPos of CellPosition
```

And with that, our move functions now have different types and can't be mixed up:

```
type PlayerXMoves = GameState * PlayerXPos -> GameState
type PlayerOMoves = GameState * PlayerOPos -> GameState
```

What is the GameState?

Now let's focus on the game state. What information do we need to represent the game completely between moves?

I think it is obvious that the only thing we need is a list of the cells, so we can define a game state like this:

```
type GameState = {
    cells : Cell list
    }
```

But now, what do we need to define a `Cell`?

First, the cell's position. Second, whether the cell has an "X" or an "O" on it. We can therefore define a cell like this:

```
type CellState =
    | X
    | O
    | Empty

type Cell = {
    pos : CellPosition
    state : CellState
    }
```

Designing the output

What about the output? What does the UI need to know in order to update itself?

One approach is just to pass the entire game state to the UI and let the UI redisplay the whole thing from scratch. Or perhaps, to be more efficient, the UI could cache the previous state and do a diff to decide what needs to be updated.

In more complicated applications, with thousands of cells, we can be more efficient and make the UI's life easier by explicitly returning the cells that changed with each move, like this:

```
// added "ChangedCells"
type PlayerXMoves = GameState * PlayerXPos -> GameState * ChangedCells
type PlayerOMoves = GameState * PlayerOPos -> GameState * ChangedCells
```

Since Tic-Tac-Toe is a tiny game, I'm going to keep it simple and just return the game state and *not* anything like `ChangedCells` as well.

But as I said at the beginning, I want the UI to be as dumb as possible! The UI should not have to "think" -- it should be given everything it needs to know by the backend, and to just follow instructions.

As it stands, the cells can be fetched directly from the `GameState`, but I'd rather that the UI did *not* know how `GameState` is defined. So let's give the UI a function (`GetCells`, say) that can extract the cells from the `GameState`:

```
type GetCells = GameState -> Cell list
```

Another approach would be for `GetCells` to return all the cells pre-organized into a 2D grid -- that would make life even easier for the UI.

```
type GetCells = GameState -> Cell[,]
```

But now the game engine is assuming the UI is using a indexed grid. Just as the UI shouldn't know about the internals of the backend, the backend shouldn't make assumptions about how the UI works.

It's fair enough to allow the UI to share the same definition of `Cell` as the backend, so we can just give the UI a list of `Cell`s and let it display them in its own way.

Ok, the UI should have everything it needs to display the game now.

Review of the first version of the design

Great! Let's look at what we've got so far:

```
module TicTacToeDomain =

    type HorizPosition = Left | HCenter | Right
    type VertPosition = Top | VCenter | Bottom
    type CellPosition = HorizPosition * VertPosition

    type CellState =
        | X
        | O
        | Empty

    type Cell = {
        pos : CellPosition
        state : CellState
        }

    type PlayerXPos = PlayerXPos of CellPosition
    type PlayerOPos = PlayerOPos of CellPosition

    // the private game state
    type GameState = exn  // use a placeholder

    // the "use-cases"
    type InitGame = unit -> GameState
    type PlayerXMoves = GameState * PlayerXPos -> GameState
    type PlayerOMoves = GameState * PlayerOPos -> GameState

    // helper function
    type GetCells = GameState -> Cell list
```

Note that in order to make this code compile while hiding the implementation of `GameState`, I've used a generic exception class (`exn`) as a placeholder for the actual implementation of `GameState`. I could also have used `unit` or `string` instead, but `exn` is not likely to get mixed up with anything else, and will prevent it being accidentally overlooked later!

A note on tuples

Just a reminder that in this design phase, I'm going to combine all the input parameters into a single tuple rather than treat them as separate parameters.

This means that I'll write:

```
InputParam1 * InputParam2 * InputParam3 -> Result
```

rather than the more standard:

```
InputParam1 -> InputParam2 -> InputParam3 -> Result
```

I'm doing this just to make the input and output obvious. When it comes to the implementation, it's more than likely that we'll switch to the standard way, so that we can take advantage of the techniques in our functional toolbox such as partial application.

Doing a design walkthrough

At this point, with a rough design in place, I like to do a walkthrough as if it were being used for real. In a larger design, I might develop a small throwaway prototype, but in this case, the design is small enough that I can do it in my head.

So, let's pretend that we are the UI and we are given the design above. We start by calling the initialization function to get a new game:

```
type InitGame = unit -> GameState
```

Ok, so now we have a `GameState` and we are ready to display the initial grid.

At this point, the UI would create, say, a grid of empty buttons, associate a cell to each button, and then draw the cell in the "empty" state.

This is fine, because the UI doesn't have to think. We are explicitly giving the UI a list of all cells, and also making the initial cell state `Empty`, so the UI doesn't have to know which is the default state -- it just displays what it is given.

One thing though. Since there is no input needed to set up the game, *and* the game state is immutable, we will have exactly the same initial state for every game.

Therefore we don't need a function to create the initial game state, just a "constant" that gets reused for each game.

```
type InitialGameState = GameState
```

When does the game stop?

Next in our walkthrough, let's play a move.

- A player, "X" or "O", clicks on a cell
- We combine the player and `CellPosition` into the appropriate type, such as a `PlayerXPos`

- We then pass that and the `GameState` into the appropriate `Move` function

```
type PlayerXMoves =
    GameState * PlayerXPos -> GameState
```

The output is a new `GameState`. The UI then calls `GetCells` to get the new cells. We loop through this list, update the display, and now we're ready to try again.

Excellent!

Umm... except for the bit about knowing when to stop.

As designed, This game will go on forever. We need to include something in the output of the move to let us know whether the game is over!

So let's create a `GameStatus` type to keep track of that.

```
type GameStatus =
    | InProcess
    | PlayerXWon
    | PlayerOWon
    | Tie
```

And we need to add it to the output of the move as well, so now we have:

```
type PlayerXMoves =
    GameState * PlayerXPos -> GameState * GameStatus
```

So now we can keep playing moves repeatedly while `GameStatus` is `InProcess` and then stop.

The pseudocode for the UI would look like

```
// loop while game not over
let rec playMove gameState =
    let pos = // get position from user input
    let newGameState,status =
        playerXMoves (gameState,pos) // process move
    match status with
    | InProcess ->
        // play another move
        playMove newGameState
    | PlayerXWon ->
        // show that player X won
    | etc

// start the game with the initial state
let startGame() =
    playMove initialGameState
```

I think we've got everything we need to play a game now, so let's move on to error handling.

What kind of errors can happen?

Before we starting thinking about the internals of the game, let's think about what kinds of errors the UI team could make when using this design:

Could the UI create an invalid `GameState` and corrupt the game?

No, because we are going to keep the internals of the game state hidden from the UI.

Could the UI pass in an invalid `CellPosition` ?

No, because the horizontal and vertical components of `CellPosition` are restricted and therefore it cannot be created in an invalid state. No validation is needed.

Could the UI pass in a *valid* `CellPosition` but at the *wrong* time?

Ah, now you're talking! Yes -- that is totally possible. In the design we have so far, there is nothing stopping a player playing the same square twice!

Could the UI allow player X to play twice in a row?

Again, yes. Nothing in our design prevents this.

What about when the game has ended but the dumb UI forgets to check the `GameStatus` and doesn't notice. Should the game logic still accept moves?

Of course not, but yet again our design fails to do this.

The big question is: can we fix these three issues *in our design* without having to rely on special validation code in the implementation? That is, can we encode these rules into *types*.

At this point you might be thinking "why bother with all these types?"

The advantage of using types over validation code is that the types are part of the design, which means that business rules like these are self-documenting. On the other hand, validation code tends to be scattered around and buried in obscure classes, so it is hard to get a big picture of all the constraints.

In general then, I prefer to use types rather than code if I can.

Enforcing the rules through types

So, can we encode these rules using types? The answer is yes!

To stop someone playing the same square twice we can change the game engine so that it outputs a list of valid moves. And then we can require that *only* items in this list are allowed to be played in the next turn.

If we do this, our move type will look like this:

```
type ValidPositionsForNextMove = CellPosition list

// a move returns the list of available positions for the next move
type PlayerXMoves =
    GameState * PlayerXPos -> // input
        GameState * GameStatus * ValidPositionsForNextMove // output
```

And we can extend this approach to stop player X playing twice in a row too. Simply make the `ValidPositionsForNextMove` be a list of `PlayerOPos` rather than generic positions. Player X will not be able to play them!

```
type ValidMovesForPlayerX = PlayerXPos list
type ValidMovesForPlayerO = PlayerOPos list

type PlayerXMoves =
    GameState * PlayerXPos -> // input
        GameState * GameStatus * ValidMovesForPlayerO // output

type PlayerOMoves =
    GameState * PlayerOPos -> // input
        GameState * GameStatus * ValidMovesForPlayerX // output
```

This approach also means that when the game is over, there are *no valid moves* available. So the UI cannot just loop forever, it will be forced to stop and deal with the situation.

So now we have encoded all three rules into the type system -- no manual validation needed.

Some refactoring

Let's do some refactoring now.

First we have a couple of choice types with a case for Player X and another similar case for Player O.

```
type CellState =
    | X
    | O
    | Empty

type GameStatus =
    | InProcess
    | PlayerXWon
    | PlayerOWon
    | Tie
```

Let's extract the players into their own type, and then we can parameterize the cases to make them look nicer:

```
type Player = PlayerO | PlayerX

type CellState =
    | Played of Player
    | Empty

type GameStatus =
    | InProcess
    | Won of Player
    | Tie
```

The second thing we can do is to note that we only need the valid moves in the `InProcess` case, not the `Won` or `Tie` cases, so let's merge `GameStatus` and `ValidMovesForPlayer` into a single type called `MoveResult`, say:

```
type ValidMovesForPlayerX = PlayerXPos list
type ValidMovesForPlayerO = PlayerOPos list

type MoveResult =
    | PlayerXToMove of GameState * ValidMovesForPlayerX
    | PlayerOToMove of GameState * ValidMovesForPlayerO
    | GameWon of GameState * Player
    | GameTied of GameState
```

We've replaced the `InProcess` case with two new cases `PlayerXToMove` and `PlayerOToMove`, which I think is actually clearer.

The move functions now look like:

```
type PlayerXMoves =
    GameState * PlayerXPos ->
        GameState * MoveResult

type PlayerOMoves =
    GameState * PlayerOPos ->
        GameState * MoveResult
```

I could have had the new `GameState` returned as part of `MoveResult` as well, but I left it "outside" to make it clear that is not to be used by the UI.

Also, leaving it outside will give us the option of writing helper code that will thread a game state through a series of calls for us. This is a more advanced technique, so I'm not going to discuss it in this post.

Finally, the `InitialGameState` should also take advantage of the `MoveResult` to return the available moves for the first player. Since it has both a game state and a initial set of moves, let's just call it `NewGame` instead.

```
type NewGame = GameState * MoveResult
```

If the initial `MoveResult` is the `PlayerXToMove` case, then we have also constrained the UI so that only player X can move first. Again, this allows the UI to be ignorant of the rules.

Second recap

So now here's the tweaked design we've got after doing the walkthrough.

```
module TicTacToeDomain =

    type HorizPosition = Left | HCenter | Right
    type VertPosition = Top | VCenter | Bottom
    type CellPosition = HorizPosition * VertPosition

    type Player = PlayerO | PlayerX

    type CellState =
        | Played of Player
        | Empty

    type Cell = {
        pos : CellPosition
        state : CellState
        }

    type PlayerXPos = PlayerXPos of CellPosition
    type PlayerOPos = PlayerOPos of CellPosition

    // the private game state
    type GameState = exn  // use a placeholder

    type ValidMovesForPlayerX = PlayerXPos list
    type ValidMovesForPlayerO = PlayerOPos list

    // the move result
    type MoveResult =
        | PlayerXToMove of ValidMovesForPlayerX
        | PlayerOToMove of ValidMovesForPlayerO
        | GameWon of Player
        | GameTied

    // the "use-cases"
    type NewGame =
        GameState * MoveResult
    type PlayerXMoves =
        GameState * PlayerXPos -> GameState * MoveResult
    type PlayerOMoves =
        GameState * PlayerOPos -> GameState * MoveResult

    // helper function
    type GetCells = GameState -> Cell list
```

We're not quite done with the outside-in design yet. One question is yet to be resolved: how can we hide the implementation of `GameState` from the UI?

Decoupling shared and private types

In any design, we want to decouple the "interface" from the "implementation". In this case, we have:

- A set of shared data structures and functions that are used by both the UI and the game engine. (`CellState` , `MoveResult` , `PlayerXPos` , etc.)
- A set of private data structures and functions that should only be accessed by the game logic. (just `GameState` so far)

It's obviously a good idea to keep these types separate. How should we do this?

In F#, the easiest way is to put them into separate modules, like this:

```
/// Types shared by the UI and the game logic
module TicTacToeDomain =

    type HorizPosition = Left | HCenter | Right
    type VertPosition = Top | VCenter | Bottom
    type CellPosition = HorizPosition * VertPosition

    type Player = PlayerO | PlayerX

    type CellState =
        | Played of Player
        | Empty

    type PlayerXMoves =
        GameState * PlayerXPos -> GameState * MoveResult
    // etc

/// Private types used by the internal game logic
module TicTacToeImplementation =
    open TicTacToeDomain

    // private implementation detail
    type GameState = {
        cells : Cell list
        }

    // etc
```

But if we want to keep the internals of the game logic private, what do we do with `GameState` ? It's used by public functions such as `PlayerXMoves` , but we want to keep its structure secret. How can we do that?

Option 1 - put the public and private types in the same module

The first choice might be to put the public and private types in the same module, and have this module be the "core" domain module that all other modules depend on.

Here's some code that demonstrates what this approach would look like:

```
module TicTacToeImplementation =

    // public types
    type HorizPosition = Left | HCenter | Right
    type VertPosition = Top | VCenter | Bottom
    type CellPosition = HorizPosition * VertPosition

    type CellState =
        | Played of Player
        | Empty

    type PlayerXMoves =
        GameState * PlayerXPos -> GameState * MoveResult
    // etc

    // --------------------
    // private types

    type private InternalType = // to do

    // --------------------
    // public types with private constructor

    type GameState = private {
        cells : Cell list
        }

    // etc
```

All the types are in one module.

Many of the types, such as `CellState`, will be public by default. That's fine.

But you can see that some of the types, such as `InternalType`, have been marked private. That means that they cannot be used outside the module at all.

Finally, `GameState` is not private, but its constructor is, which means that it *can* be used outside the module, but because its constructor is private, new ones can't be created, which sounds like what we need.

We might have appeared to solve the issue, but this approach often causes problems of its own. For starters, trying to keep the `public` and `private` qualifiers straight can cause annoying compiler errors, such as this one:

```
The type 'XXX' is less accessible than the value, member or type 'YYY' it is used in
```

And even if this weren't a problem, putting the "interface" and the "implementation" in the same file will generally end up creating extra complexity as the implementation gets larger.

Option 2 - representing `GameState` with an abstract base class

The object-oriented way of approaching this would be to represent `GameState` as an abstract base class or interface, and then have a particular implementation inherit from the abstract class.

This allows all the shared types to reference the abstract class or interface safely, while any particular implementation is hidden.

Here's how you might do this in F#:

```
/// Types shared by the UI and the game logic
module TicTacToeDomain =

    // abstract base class
    type GameState() = class end

/// Private types used by the internal game logic
module TicTacToeImplementation =
    open TicTacToeDomain

    type GameStateImpl() =
        inherit GameState()

    // etc
```

But alas, there are problems with this approach too.

First, it's not very functional, is it? F# does support classes and interfaces for those situations when we need them, but we should really be able to find a more idiomatic functional solution than this!

Second, it's potentially not safe. The actual implementation would have to downcast `GameState` into the type it expects in order to get at the internal data. But if I had *two* implementations that inherited `GameState`, what's to stop me passing a game state from implementation B into a function that is expecting a game state from implementation A? Nothing! Chaos would ensue!

Note that in a pure OO model this situation could not happen because the `GameState` itself would have stateful methods instead of the pure functional API that we have here.

Option 3 - parameterize the implementation

Let's think about the requirements again: "The `GameState` is public but we don't know what the implementation will be."

When you rephrase it like this, the functional way of modeling this becomes clear, which is to use *generic parameters* (aka "parametric polymorphism").

In other words, we make `GameState` a *generic type* which represents a particular implementation.

This means that the UI can work with the `GameState` type, but because the actual implementation type used is not known, the UI cannot accidentally "look inside" and extract any information, *even if the implementation type is public*.

This last point is important, so I'm going to say it again with another example. If I give you a object of type `List<T>` in C#, you can work with the list in many ways, but you cannot know what the `T` is, and so you can never accidentally write code that assumes that `T` is an `int` or a `string` or a `bool`. And this "hidden-ness" has got nothing to do with whether `T` is a public type or not.

If we do take this approach then we can allow the internals of the game state to be completely public, safe in the knowledge that the UI cannot use that information even if it wanted to!

So here's some code demonstrating this approach.

First the shared types, with `GameState<'T>` being the parameterized version.

```
/// Types shared by the UI and the game logic
module TicTacToeDomain =

    // unparameterized types
    type PlayerXPos = PlayerXPos of CellPosition
    type PlayerOPos = PlayerOPos of CellPosition

    // parameterized types
    type PlayerXMoves<'GameState> =
        'GameState * PlayerXPos -> 'GameState * MoveResult
    type PlayerOMoves<'GameState> =
        'GameState * PlayerOPos -> 'GameState * MoveResult

    // etc
```

The types that don't use the game state are unchanged, but you can see that `PlayerXMoves<'T>` has been parameterized with the game state.

Adding generics like this can often cause cascading changes to many types, forcing them all to be parameterized. Dealing with all these generics is one reason why type inference is so helpful in practice!

Now for the types internal to the game logic. They can all be public now, because the UI won't be able to know about them.

```
module TicTacToeImplementation =
    open TicTacToeDomain

    // can be public
    type GameState = {
        cells : Cell list
        }
```

Finally, here's what the implementation of a `playerXMoves` function might look like:

```
let playerXMoves : PlayerXMoves<GameState> =
    fun (gameState,move) ->
        // logic
```

This function references a particular implementation, but can be passed into the UI code because it conforms to the `PlayerXMoves<'T>` type.

Furthermore, by using generic parameters, we naturally enforce that the same implementation, say "GameStateA", is used throughout.

In other words, the game state created by `InitGame<GameStateA>` can *only* be passed to a `PlayerXMoves<GameStateA>` function which is parameterized on the *same* implementation type.

Glueing it all together with "dependency injection"

Finally, let's talk about how everything can be glued together.

The UI code will be designed to work with a *generic* implementation of `GameState`, and thus generic versions of the `newGame` and `move` functions.

But of course, at some point we need to get access to the `newGame` and `move` functions for a *specific* implementation. What's the best way to glue all this together?

The answer is the functional equivalent of dependency injection. We will have an "application" or "program" component as a top-level layer that will construct an implementation and pass it to the UI.

Here's an example of what such code would look like:

- The `GameImplementation` module exports specific implementations of `newGame` and the `move` functions.
- The `UserInterface` module exports a `TicTacToeForm` class that accepts these implementations in its constructor.
- The `Application` module glues everything together. It creates a `TicTacToeForm` and passes it the implementations exported from the `GameImplementation` module.

Here's some code to demonstrate this approach:

```
module TicTacToeImplementation =
    open TicTacToeDomain

    /// create the state of a new game
    let newGame : NewGame<GameState> =
        // return new game and current available moves
        let validMoves = // to do
        gameState, PlayerXToMove validMoves

    let playerXMoves : PlayerXMoves<GameState> =
        fun (gameState,move) ->
            // implementation

module WinFormUI =
    open TicTacToeDomain
    open System.Windows.Forms

    type TicTacToeForm<'T>
        (
        // pass in the required functions
        // as parameters to the constructor
        newGame:NewGame<'T>,
        playerXMoves:PlayerXMoves<'T>,
        playerOMoves:PlayerOMoves<'T>,
        getCells:GetCells<'T>
        ) =
        inherit Form()
     // implementation to do

module WinFormApplication =
    open WinFormUI

    // get functions from implementation
    let newGame = TicTacToeImplementation.newGame
    let playerXMoves = TicTacToeImplementation.playerXMoves
    let playerOMoves = TicTacToeImplementation.playerOMoves
    let getCells = TicTacToeImplementation.getCells

    // create form and start game
    let form =
        new TicTacToeForm<_>(newGame,playerXMoves,playerOMoves,getCells)
    form.Show()
```

A few notes on this code:

First, I'm using WinForms rather than WPF because it has Mono support and because it works without NuGet dependencies. If you want to use something better, check out ETO.Forms.

Next, you can see that I've explicitly added the type parameters to `TicTacToeForm<'T>` like this.

```
TicTacToeForm<'T>(newGame:NewGame<'T>, playerXMoves:PlayerXMoves<'T>, etc)
```

I could have eliminated the type parameter for the form by doing something like this instead:

```
TicTacToeForm(newGame:NewGame<_>, playerXMoves:PlayerXMoves<_>, etc)
```

or even:

```
TicTacToeForm(newGame, playerXMoves, etc)
```

and let the compiler infer the types, but this often causes a "less generic" warning like this:

```
warning FS0064: This construct causes code to be less generic than indicated by the ty
pe annotations.
The type variable 'T has been constrained to be type 'XXX'.
```

By explicitly writing `TicTacToeForm<'T>`, this can be avoided, although it is ugly for sure.

Some more refactoring

We've got four different functions to export. That's getting a bit much so let's create a record to store them in:

```
// the functions exported from the implementation
// for the UI to use.
type TicTacToeAPI<'GameState>  =
    {
    newGame : NewGame<'GameState>
    playerXMoves : PlayerXMoves<'GameState>
    playerOMoves : PlayerOMoves<'GameState>
    getCells : GetCells<'GameState>
    }
```

This acts both as a container to pass around functions, *and* as nice documentation of what functions are available in the API.

The implementation now has to create an "api" object:

```
module TicTacToeImplementation =
    open TicTacToeDomain

    /// create the functions to export
    let newGame : NewGame<GameState> = // etc
    let playerXMoves : PlayerXMoves<GameState> = // etc
    // etc

    // export the functions
    let api = {
        newGame = newGame
        playerOMoves = playerOMoves
        playerXMoves = playerXMoves
        getCells = getCells
        }
```

But the UI code simplifies as a result:

```
module WinFormUI =
    open TicTacToeDomain
    open System.Windows.Forms

    type TicTacToeForm<'T>(api:TicTacToeAPI<'T>) =
        inherit Form()
     // implementation to do

module WinFormApplication =
    open WinFormUI

    // get functions from implementation
    let api = TicTacToeImplementation.api

    // create form and start game
    let form = new TicTacToeForm<_>(api)
    form.Show()
```

Prototyping a minimal implementation

It seems like we're getting close to a final version. But let's do one more walkthrough to exercise the "dependency injection" design, this time writing some minimal code to test the interactions.

For example, here is some minimal code to implement the `newGame` and `playerXMoves` functions.

- The `newGame` is just an game with no cells and no available moves
- The minimal implementation of `move` is easy -- just return game over!

```
let newGame : NewGame<GameState> =
    // create initial game state with empty everything
    let gameState = { cells=[]}
    let validMoves = []
    gameState, PlayerXToMove validMoves

let playerXMoves : PlayerXMoves<GameState> =
    // dummy implementation
    fun gameState move ->  gameState,GameTied

let playerOMoves : PlayerOMoves<GameState> =
    // dummy implementation
    fun gameState move ->  gameState,GameTied

let getCells gameState =
    gameState.cells

let api = {
    newGame = newGame
    playerOMoves = playerOMoves
    playerXMoves = playerXMoves
    getCells = getCells
    }
```

Now let's create a minimal implementation of the UI. We won't draw anything or respond to clicks, just mock up some functions so that we can test the logic.

Here's my first attempt:

```
type TicTacToeForm<'GameState>(api:TicTacToeAPI<'GameState>) =
    inherit Form()

    let mutable gameState : 'GameState = ???
    let mutable lastMoveResult : MoveResult = ???

    let displayCells gameState =
        let cells = api.getCells gameState
        for cell in cells do
            // update display

    let startGame()=
        let initialGameState,initialResult = api.newGame
        gameState <- initialGameState
        lastMoveResult <- initialResult
        // create cell grid from gameState

    let handleMoveResult moveResult =
        match moveResult with
        | PlayerXToMove availableMoves ->
            // show available moves
```

```
        | PlayerOToMove availableMoves ->
            // show available moves
        | GameWon player ->
            let msg = sprintf "%A Won" player
            MessageBox.Show(msg) |> ignore
        | GameTied ->
            MessageBox.Show("Tied") |> ignore

    // handle a click
    let handleClick() =
        let gridIndex = 0,0  // dummy for now
        let cellPos = createCellPosition gridIndex
        match lastMoveResult with
        | PlayerXToMove availableMoves ->
            let playerXmove = PlayerXPos cellPos
            // if move is in available moves then send it
            // to the api
            let newGameState,newResult =
                api.playerXMoves gameState playerXmove
            handleMoveResult newResult

            //update the globals
            gameState <- newGameState
            lastMoveResult <- newResult
        | PlayerOToMove availableMoves ->
            let playerOmove = PlayerOPos cellPos
            // if move is in available moves then send it
            // to the api
            // etc
        | GameWon player ->
            ?? // we aleady showed after the last move
```

As you can see, I'm planning to use the standard Form event handling approach -- each cell will have a "clicked" event handler associated with it. How the control or pixel location is converted to a `CellPosition` is something I'm not going to worry about right now, so I've just hard-coded some dummy data.

I'm *not* going to be pure here and have a recursive loop. Instead, I'll keep the current `gameState` as a mutable which gets updated after each move.

But now we have got a tricky situation... What is the `gameState` when the game hasn't started yet? What should we initialize it to? Similarly, when the game is over, what should it be set to?

```
let mutable gameState : 'GameState = ???
```

One choice might be to use a `GameState option` but that seems like a hack, and it makes me think that we are failing to think of something.

Similarly, we have a field to hold the result of the last move (`lastMoveResult`) so we can keep track of whose turn it is, or whether the game is over.

But again, what should it be set to when the game hasn't started?

Let's take a step back and look at all the states the user interface can be in -- not the state of the *game* itself, but the state of the *user interface*.

- We start off in an "idle" state, with no game being played.
- Then the user starts the game, and we are "playing".
- While each move is played, we stay in the "playing" state.
- When the game is over, we show the win or lose message.
- We wait for the user to acknowledge the end-of-game message, then go back to idle again.

Again, this is for the UI only, it has nothing to do with the internal game state.

So -- our solution to all problems! -- let's create a type to represent these states.

```
type UiState =
    | Idle
    | Playing
    | Won
    | Lost
```

But do we really need the `Won` and `Lost` states? Why don't we just go straight back to `Idle` when the game is over?

So now the type looks like this:

```
type UiState =
    | Idle
    | Playing
```

The nice thing about using a type like this is that we can easily store the data that we need for each state.

- What data do we need to store in the `Idle` state? Nothing that I can think of.
- What data do we need to store in the `Playing` state? Well, this would be a perfect place to keep track of the `gameState` and `lastMoveResult` that we were having problems with earlier. They're only needed when the game is being played, but not otherwise.

So our final version looks like this. We've had to add the `<'GameState>` to `UiState` because we don't know what the actual game state is.

```
type UiState<'GameState> =
    | Idle
    | Playing of 'GameState * MoveResult
```

With this type now available, we no longer need to store the game state directly as a field in the class. Instead we store a mutable `UiState`, which is initialized to `Idle`.

```
type TicTacToeForm<'GameState>(api:TicTacToeAPI<'GameState>) =
    inherit Form()

    let mutable uiState = Idle
```

When we start the game, we change the UI state to be `Playing`:

```
let startGame()=
    uiState <- Playing api.newGame
    // create cell grid from gameState
```

And when we handle a click, we only do something if the uiState is in `Playing` mode, and then we have no trouble getting the `gameState` and `lastMoveResult` that we need, because it is stored as part of the data for that case.

```
let handleClick() =
    match uiState with
    | Idle -> ()
        // do nothing

    | Playing (gameState,lastMoveResult) ->
        let gridIndex = 0,0  // dummy for now
        let cellPos = createCellPosition gridIndex
        match lastMoveResult with
        | PlayerXToMove availableMoves ->
            let playerXmove = PlayerXPos cellPos
            // if move is in available moves then send it
            // to the api
            let newGameState,newResult =
                api.playerXMoves gameState playerXmove

            // handle the result
            // e.g. if the game is over
            handleMoveResult newResult

            // update the uiState with newGameState
            uiState <- Playing (newGameState,newResult)

        | PlayerOToMove availableMoves ->
            // etc
        | _ ->
            // ignore other states
```

If you look at the last line of the `PlayerXToMove` case, you can see the global `uiState` field being updated with the new game state:

```
| PlayerXToMove availableMoves ->
    // snipped

    let newGameState,newResult = // get new state

    // update the uiState with newGameState
    uiState <- Playing (newGameState,newResult)
```

So where have we got to with this bit of prototyping?

It's pretty ugly, but it has served its purpose.

The goal was to quickly implement the UI to see if the design held up in use, and I think we can say that it did, because the design of the domain types and api has remained unchanged.

We also understand the UI requirements a bit better, which is a bonus. I think we can stop now!

The complete game, part 1: The design

To finish off, I'll show the code for the complete game, including implementation and user interface.

If you don't want to read this code, you can skip to the questions and summary below.

All the code shown is available on GitHub in this gist.

We'll start with our final domain design:

```
module TicTacToeDomain =

    type HorizPosition = Left | HCenter | Right
    type VertPosition = Top | VCenter | Bottom
    type CellPosition = HorizPosition * VertPosition

    type Player = PlayerO | PlayerX

    type CellState =
        | Played of Player
        | Empty

    type Cell = {
        pos : CellPosition
        state : CellState
        }

    type PlayerXPos = PlayerXPos of CellPosition
    type PlayerOPos = PlayerOPos of CellPosition

    type ValidMovesForPlayerX = PlayerXPos list
    type ValidMovesForPlayerO = PlayerOPos list

    type MoveResult =
        | PlayerXToMove of ValidMovesForPlayerX
        | PlayerOToMove of ValidMovesForPlayerO
        | GameWon of Player
        | GameTied

    // the "use-cases"
    type NewGame<'GameState> =
        'GameState * MoveResult
    type PlayerXMoves<'GameState> =
        'GameState -> PlayerXPos -> 'GameState * MoveResult
    type PlayerOMoves<'GameState> =
        'GameState -> PlayerOPos -> 'GameState * MoveResult

    // helper function
    type GetCells<'GameState> =
        'GameState -> Cell list

    // the functions exported from the implementation
    // for the UI to use.
    type TicTacToeAPI<'GameState>  =
        {
        newGame : NewGame<'GameState>
        playerXMoves : PlayerXMoves<'GameState>
        playerOMoves : PlayerOMoves<'GameState>
        getCells : GetCells<'GameState>
        }
```

The complete game, part 2: The game logic implementation

Next, here's a complete implementation of the design which I will not discuss in detail. I hope that the comments are self-explanatory.

```
module TicTacToeImplementation =
    open TicTacToeDomain

    /// private implementation of game state
    type GameState = {
        cells : Cell list
        }

    /// the list of all horizontal positions
    let allHorizPositions = [Left; HCenter; Right]

    /// the list of all horizontal positions
    let allVertPositions = [Top; VCenter; Bottom]

    /// A type to store the list of cell positions in a line
    type Line = Line of CellPosition list

    /// a list of the eight lines to check for 3 in a row
    let linesToCheck =
        let makeHLine v = Line [for h in allHorizPositions do yield (h,v)]
        let hLines= [for v in allVertPositions do yield makeHLine v]

        let makeVLine h = Line [for v in allVertPositions do yield (h,v)]
        let vLines = [for h in allHorizPositions do yield makeVLine h]

        let diagonalLine1 = Line [Left,Top; HCenter,VCenter; Right,Bottom]
        let diagonalLine2 = Line [Left,Bottom; HCenter,VCenter; Right,Top]

        // return all the lines to check
        [
        yield! hLines
        yield! vLines
        yield diagonalLine1
        yield diagonalLine2
        ]

    /// get the cells from the gameState
    let getCells gameState =
        gameState.cells

    /// get the cell corresponding to the cell position
    let getCell gameState posToFind =
        gameState.cells
        |> List.find (fun cell -> cell.pos = posToFind)
```

```
/// update a particular cell in the GameState
/// and return a new GameState
let private updateCell newCell gameState =

    // create a helper function
    let substituteNewCell oldCell =
        if oldCell.pos = newCell.pos then
            newCell
        else
            oldCell

    // get a copy of the cells, with the new cell swapped in
    let newCells = gameState.cells |> List.map substituteNewCell

    // return a new game state with the new cells
    {gameState with cells = newCells }

/// Return true if the game was won by the specified player
let private isGameWonBy player gameState =

    // helper to check if a cell was played by a particular player
    let cellWasPlayedBy playerToCompare cell =
        match cell.state with
        | Played player -> player = playerToCompare
        | Empty -> false

    // helper to see if every cell in the Line has been played by the same player
    let lineIsAllSamePlayer player (Line cellPosList) =
        cellPosList
        |> List.map (getCell gameState)
        |> List.forall (cellWasPlayedBy player)

    linesToCheck
    |> List.exists (lineIsAllSamePlayer player)


/// Return true if all cells have been played
let private isGameTied gameState =
    // helper to check if a cell was played by any player
    let cellWasPlayed cell =
        match cell.state with
        | Played _ -> true
        | Empty -> false

    gameState.cells
    |> List.forall cellWasPlayed

/// determine the remaining moves for a player
let private remainingMovesForPlayer playerMove gameState =

    // helper to return Some if a cell is playable
    let playableCell cell =
```

```
            match cell.state with
            | Played player -> None
            | Empty -> Some (playerMove cell.pos)

        gameState.cells
        |> List.choose playableCell


    /// create the state of a new game
    let newGame =

        // allPositions is the cross-product of the positions
        let allPositions = [
            for h in allHorizPositions do
            for v in allVertPositions do
                yield (h,v)
            ]

        // all cells are empty initially
        let emptyCells =
            allPositions
            |> List.map (fun pos -> {pos = pos; state = Empty})

        // create initial game state
        let gameState = { cells=emptyCells }

        // initial set of valid moves for player X is all positions
        let validMoves =
            allPositions
            |> List.map PlayerXPos

        // return new game
        gameState, PlayerXToMove validMoves

    // player X makes a move
    let playerXMoves gameState (PlayerXPos cellPos) =
        let newCell = {pos = cellPos; state = Played PlayerX}
        let newGameState = gameState |> updateCell newCell

        if newGameState |> isGameWonBy PlayerX then
            // return the new state and the move result
            newGameState, GameWon PlayerX
        elif newGameState |> isGameTied then
            // return the new state and the move result
            newGameState, GameTied
        else
            let remainingMoves =
                newGameState |> remainingMovesForPlayer PlayerOPos
            newGameState, PlayerOToMove remainingMoves

    // player O makes a move
    let playerOMoves gameState (PlayerOPos cellPos) =
        let newCell = {pos = cellPos; state = Played PlayerO}
```

```
        let newGameState = gameState |> updateCell newCell

        if newGameState |> isGameWonBy PlayerO then
            // return the new state and the move result
            newGameState, GameWon PlayerO
        elif newGameState |> isGameTied then
            // return the new state and the move result
            newGameState, GameTied
        else
            let remainingMoves =
                newGameState |> remainingMovesForPlayer PlayerXPos
            newGameState, PlayerXToMove remainingMoves

        // Exercise - refactor to remove the duplicate code from
        // playerXMoves  and playerOMoves


    /// export the API to the application
    let api = {
        newGame = newGame
        playerOMoves = playerOMoves
        playerXMoves = playerXMoves
        getCells = getCells
        }
```

The complete game, part 3: A console based user interface

And to complete the implementation, here's the code for a console based user interface.

Obviously this part of the implementation is not pure! I'm writing to and reading from the console, duh. If you want to be extra good, it would be easy enough to convert this to a pure implementation using `IO` or similar.

Personally, I like to focus on the core domain logic being pure and I generally don't bother about the UI too much, but that's just me.

```
/// Console based user interface
module ConsoleUi =
    open TicTacToeDomain

    /// Track the UI state
    type UserAction<'a> =
        | ContinuePlay of 'a
        | ExitGame

    /// Print each available move on the console
    let displayAvailableMoves moves =
```

```
        moves
        |> List.iteri (fun i move ->
            printfn "%i) %A" i move )

    /// Get the move corresponding to the
    /// index selected by the user
    let getMove moveIndex moves =
        if moveIndex < List.length moves then
            let move = List.nth moves moveIndex
            Some move
        else
            None

    /// Given that the user has not quit, attempt to parse
    /// the input text into a index and then find the move
    /// corresponding to that index
    let processMoveIndex inputStr gameState availableMoves makeMove processInputAgain
 =
        match Int32.TryParse inputStr with
        // TryParse will output a tuple (parsed?,int)
        | true,inputIndex ->
            // parsed ok, now try to find the corresponding move
            match getMove inputIndex availableMoves with
            | Some move ->
                // corresponding move found, so make a move
                let moveResult = makeMove gameState move
                ContinuePlay moveResult // return it
            | None ->
                // no corresponding move found
                printfn "...No move found for inputIndex %i. Try again" inputIndex
                // try again
                processInputAgain()
        | false, _ ->
            // int was not parsed
            printfn "...Please enter an int corresponding to a displayed move."

            // try again
            processInputAgain()

    /// Ask the user for input. Process the string entered as
    /// a move index or a "quit" command
    let rec processInput gameState availableMoves makeMove =

        // helper that calls this function again with exactly
        // the same parameters
        let processInputAgain() =
            processInput gameState availableMoves makeMove

        printfn "Enter an int corresponding to a displayed move or q to quit:"
        let inputStr = Console.ReadLine()
        if inputStr = "q" then
            ExitGame
        else
```

```
            processMoveIndex inputStr gameState availableMoves makeMove processInputAg
ain

    /// Display the cells on the console in a grid
    let displayCells cells =
        let cellToStr cell =
            match cell.state with
            | Empty -> "-"
            | Played player ->
                match player with
                | PlayerO -> "O"
                | PlayerX -> "X"

        let printCells cells  =
            cells
            |> List.map cellToStr
            |> List.reduce (fun s1 s2 -> s1 + "|" + s2)
            |> printfn "|%s|"

        let topCells =
            cells |> List.filter (fun cell -> snd cell.pos = Top)
        let centerCells =
            cells |> List.filter (fun cell -> snd cell.pos = VCenter)
        let bottomCells =
            cells |> List.filter (fun cell -> snd cell.pos = Bottom)

        printCells topCells
        printCells centerCells
        printCells bottomCells
        printfn ""   // add some space

    /// After each game is finished,
    /// ask whether to play again.
    let rec askToPlayAgain api  =
        printfn "Would you like to play again (y/n)?"
        match Console.ReadLine() with
        | "y" ->
            ContinuePlay api.newGame
        | "n" ->
            ExitGame
        | _ -> askToPlayAgain api

    /// The main game loop, repeated
    /// for each user input
    let rec gameLoop api userAction =
        printfn "\n------------------------------\n"  // a separator between moves

        match userAction with
        | ExitGame ->
            printfn "Exiting game."
        | ContinuePlay (state,moveResult) ->
            // first, update the display
            state |> api.getCells |> displayCells
```

```
            // then handle each case of the result
            match moveResult with
            | GameTied ->
                printfn "GAME OVER - Tie"
                printfn ""
                let nextUserAction = askToPlayAgain api
                gameLoop api nextUserAction
            | GameWon player ->
                printfn "GAME WON by %A" player
                printfn ""
                let nextUserAction = askToPlayAgain api
                gameLoop api nextUserAction
            | PlayerOToMove availableMoves ->
                printfn "Player O to move"
                displayAvailableMoves availableMoves
                let newResult = processInput state availableMoves api.playerOMoves
                gameLoop api newResult
            | PlayerXToMove availableMoves ->
                printfn "Player X to move"
                displayAvailableMoves availableMoves
                let newResult = processInput state availableMoves api.playerXMoves
                gameLoop api newResult

    /// start the game with the given API
    let startGame api =
        let userAction = ContinuePlay api.newGame
        gameLoop api userAction
```

And finally, the application code that connects all the components together and launches the UI:

```
module ConsoleApplication =

    let startGame() =
        let api = TicTacToeImplementation.api
        ConsoleUi.startGame api
```

Example game

Here's what the output of this game looks like:

```
|-|X|-|
|X|-|-|
|0|-|-|

Player O to move
0) PlayerOPos (Left, Top)
```

```
1) PlayerOPos (HCenter, VCenter)
2) PlayerOPos (HCenter, Bottom)
3) PlayerOPos (Right, Top)
4) PlayerOPos (Right, VCenter)
5) PlayerOPos (Right, Bottom)
Enter an int corresponding to a displayed move or q to quit:
1

------------------------------

|-|X|-|
|X|O|-|
|O|-|-|

Player X to move
0) PlayerXPos (Left, Top)
1) PlayerXPos (HCenter, Bottom)
2) PlayerXPos (Right, Top)
3) PlayerXPos (Right, VCenter)
4) PlayerXPos (Right, Bottom)
Enter an int corresponding to a displayed move or q to quit:
1

------------------------------

|-|X|-|
|X|O|-|
|O|X|-|

Player O to move
0) PlayerOPos (Left, Top)
1) PlayerOPos (Right, Top)
2) PlayerOPos (Right, VCenter)
3) PlayerOPos (Right, Bottom)
Enter an int corresponding to a displayed move or q to quit:
1

------------------------------

|-|X|O|
|X|O|-|
|O|X|-|

GAME WON by PlayerO

Would you like to play again (y/n)?
```

Logging

Oops! We promised we would add logging to make it enterprise-ready!

That's easy -- all we have to do is replace the api functions with equivalent functions that log the data we're interested in

```
module Logger =
    open TicTacToeDomain

    let logXMove (PlayerXPos cellPos)=
        printfn "X played %A" cellPos

    let logOMove (PlayerOPos cellPos)=
        printfn "O played %A" cellPos

    /// inject logging into the API
    let injectLogging api =

        // make a logged version of the game function
        let playerXMoves state move =
            logXMove move
            api.playerXMoves state move

        // make a logged version of the game function
        let playerOMoves state move =
            logOMove move
            api.playerOMoves state move

        // create a new API with
        // the move functions replaced
        // with logged versions
        { api with
            playerXMoves = playerXMoves
            playerOMoves = playerOMoves
            }
```

Obviously, in a real system you'd replace it with a proper logging tool such as `log4net` and generate better output, but I think this demonstrates the idea.

Now to use this, all we have to do is change the top level application to transform the original api to a logged version of the api:

```
module ConsoleApplication =

    let startGame() =
        let api = TicTacToeImplementation.api
        let loggedApi = Logger.injectLogging api
        ConsoleUi.startGame loggedApi
```

And that's it. Logging done!

Oh, and remember that I originally had the initial state created as a function rather than as a constant?

```
type InitGame = unit -> GameState
```

I changed to a constant early on in the design. But I'm regretting that now, because it means that I can't hook into the "init game" event and log it. If I do want to log the start of each game, I should really change it back to a function again.

Questions

Question: You went to the trouble of hiding the internal structure of `GameState`, yet the `PlayerXPos` and `PlayerOPos` types are public. Why?

I forgot! And then laziness kept me from updating the code, since this is really just an exercise in design.

It's true that in the current design, a malicious user interface could construct a `PlayerXPos` and then play X when it is not player X's turn, or to play a position that has already been played.

You could prevent this by hiding the implementation of `PlayerXPos` in the same way as we did for game state, using a type parameter. And of course you'd have to tweak all the related classes too.

Here's a snippet of what that would look like:

```
type MoveResult<'PlayerXPos,'PlayerOPos> =
    | PlayerXToMove of 'PlayerXPos list
    | PlayerOToMove of 'PlayerOPos list
    | GameWon of Player
    | GameTied

type NewGame<'GameState,'PlayerXPos,'PlayerOPos> =
    'GameState * MoveResult<'PlayerXPos,'PlayerOPos>

type PlayerXMoves<'GameState,'PlayerXPos,'PlayerOPos> =
    'GameState -> 'PlayerXPos ->
        'GameState * MoveResult<'PlayerXPos,'PlayerOPos>
type PlayerOMoves<'GameState,'PlayerXPos,'PlayerOPos> =
    'GameState -> 'PlayerOPos ->
        'GameState * MoveResult<'PlayerXPos,'PlayerOPos>
```

We'd also need a way for the UI to see if the `CellPosition` a user selected was valid. That is, given a `MoveResult` and the desired `CellPosition`, if the position *is* valid, return `Some` move, otherwise return `None`.

```
type GetValidXPos<'PlayerXPos,'PlayerOPos> =
    CellPosition * MoveResult<'PlayerXPos,'PlayerOPos> -> 'PlayerXPos option
```

It's getting kind of ugly now, though. That's one problem with type-first design: the type parameters can get complicated!

So it's a trade-off. How much do you use types to prevent accidental bugs without overwhelming the design?

In this case, I do think the `GameState` should be secret, as it is likely to change in the future and we want to ensure that the UI is not accidentally coupled to implementation details.

For the move types though, (a) I don't see the implementation changing and (b) the consequence of a malicious UI action is not very high, so overall I don't mind having the implementation be public.

UPDATE 2015-02-16: In the next post I solve this problem in a more elegant way, and get rid of `GameState` as well!

Question: Why are you using that strange syntax for defining the `initGame` and `move` functions?

You mean, why I am defining the functions like this:

```
/// create the state of a new game
let newGame : NewGame<GameState> =
    // implementation

let playerXMoves : PlayerXMoves<GameState> =
    fun (gameState,move) ->
        // implementation
```

rather than in the "normal" way like this:

```
/// create the state of a new game
let newGame  =
    // implementation

let playerXMoves (gameState,move) =
    // implementation
```

I'm doing this when I want to treat functions as values. Just as we might say "*x* is a value of type *int*" like this `x :int = ...`, I'm saying that "*playerXMoves* is a value of type *PlayerXMoves*" like this: `playerXMoves : PlayerXMoves = ...`. It's just that in this case, the value is a function rather than a simple value.

Doing it this way follows from the type-first approach: create a type, then implement things that conform to that type.

Would I recommend doing this for normal code? Probably not!

I'm only doing this as part of an exploratory design process. Once the design stabilizes, I would tend to switch back to the normal way of doing things.

Question: This seems like a lot of work. Isn't this just BDUF under another guise?

This might seem like quite a long winded way of doing design, but in practice, it would probably not take very long. Certainly no longer than mocking up an exploratory prototype in another language.

We've gone through a number of quick iterations, using types to document the design, and using the REPL as a "executable spec checker" to make sure that it all works together properly.

And at the end of all this, we now have a decent design with some nice properties:

- There is a "API" that separates the UI from the core logic, so that work on each part can proceed in parallel if needed.
- The types act as documentation and will constrain the implementation in a way that UML diagrams could never do!
- The design is encoded in types, so that ~~any~~ the inevitable changes that occur during development can be made with confidence.

I think this whole process is actually pretty agile, once you get used to working this way.

Question: Come on, would you *really* write Tic-Tac-Toe this way?

It depends. If it was just me, maybe not. :-)

But if it was a more complex system with different teams for the front-end and back-end, then I would certainly use a design-first approach like this. In cases like that, things like data-hiding and abstract interfaces are critical, and I think this approach delivers that.

Question: Why is the design so specific? It seems like none of it will be reusable at all. Why not?

Yes, this code is full of very specific types: `Cell`, `GameState`, etc. And it's true that none of it will be reusable.

There is always a tension between a very domain-specific and non-reusable design, like this one, and an abstract and reusable library of things like lists and trees.

Ideally, you would start with low-level, reusable components and then compose them into larger more-specific ones (e.g. a DSL), from which you can build a application. (Tomas has a good post on exactly this).

The reasons why I did not do that here is that, first, I always like to start with very *concrete* designs. You can't even know what a good abstraction looks like until you have built something a few times.

We have separated the UI from the core logic, but going any further than that does not make sense to me right now. If I was going to build lots of other kinds of games that were similar to Tic-Tac-Toe, then some useful abstractions might become apparent.

Second, designs with concrete types are easier for non-experts to understand. I'd like to think that I could show these domain types to a non-programmer (e.g. a domain expert) and have them understand and comment sensibly on them. If they were more abstract, that would not be possible.

Exercises

If you want a challenge, here are some exercises for you:

- The `playerXMoves` and `playerOMoves` functions have very similar code. How would you refactor them to reduce that?
- Do a security audit and think of all the ways that a malicious user or UI could corrupt the game using the current design. Then fix them!

Summary

In this post, we've seen how to design a system using mostly types, with the occasional code fragments to help us clarify issues.

It was definitely an exercise in design overkill but I hope that there are some ideas in there that might be applicable to real non-toy projects.

At the start, I claimed that this design would be "enterprise" ready. Is it?

- We *do* have separation of concerns via the functions that are exported to the UI.
- We *do* have a well documented API. There are no magic numbers, the names of the types are self-documenting, and the list of functions exported is in one place.
- We *do* have a security model to prevent unauthorized actions from occurring. As it

stands, it would be hard to accidentally mess up. And if we go the extra distance by parameterizing the move types as well, then it becomes really quite hard for the game to be corrupted.
- We *do* have well-documented code, I think. Even though this is "enterprise", the code is quite explicit in what it does. There are no wasted abstractions -- no `AbstractSingletonProxyFactoryBean` to make fun of.
- We *did* add auditing and logging easily, and in an elegant way after the fact, without interfering with the core design.
- We get *scalability* for free because there is no global session data. All we have to do is persist the game state in the browser (Or we could use MongoDb and be web scale).

This is not a perfect design -- I can think of a number of ways to improve it -- but overall I'm quite happy with it, considering it was a straight brain-to-code dump.

What do you think? Let me know in the comments.

UPDATE 2015-02-16: I ended up being unhappy with this design after all. In the next post I tell you why, and present a better design.

NOTE: The code for this post is available on GitHub in this gist.

Enterprise Tic-Tac-Toe, part 2

UPDATE: Slides and video from my talk on this topic

This post is one of series in I which I hope to close the gap between theory and practice in functional programming. I pick a small project and show you my thought processes as I go about designing and implementing it from beginning to end.

In the previous post, I did a design for a Tic-Tac-Toe (aka Noughts and Crosses) game.

It wasn't bad for a direct-to-code brain dump, but there were a couple of things that I wasn't happy with.

Unfortunately, the more I thought about it, the more those little niggles became full-fledged annoyances, and I got unhappier and unhappier.

In this post, I'll explain why I was so unhappy, and how I arrived at a design that I am much more satisfied with.

The old design

To recap the previous post briefly, here is the old design:

- There is a hidden `GameState` known only to the implementation.
- There are some functions that allow the players to move (`PlayerXMoves` and `PlayerOMoves`).
- The UI (or other client) passes the game state into each move, and gets a updated game state back.
- Each move also returns a `MoveResult` which contains the game status (in process, won, tied), and if the game is still in process, whose turn it is, and what the available moves are.

Here's the code:

```
module TicTacToeDomain =

    type HorizPosition = Left | HCenter | Right
    type VertPosition = Top | VCenter | Bottom
    type CellPosition = HorizPosition * VertPosition

    type Player = PlayerO | PlayerX

    type CellState =
        | Played of Player
        | Empty

    type Cell = {
        pos : CellPosition
        state : CellState
        }

    type PlayerXPos = PlayerXPos of CellPosition
    type PlayerOPos = PlayerOPos of CellPosition

    type ValidMovesForPlayerX = PlayerXPos list
    type ValidMovesForPlayerO = PlayerOPos list

    type MoveResult =
        | PlayerXToMove of ValidMovesForPlayerX
        | PlayerOToMove of ValidMovesForPlayerO
        | GameWon of Player
        | GameTied

    // the "use-cases"
    type NewGame<'GameState> =
        'GameState * MoveResult
    type PlayerXMoves<'GameState> =
        'GameState -> PlayerXPos -> 'GameState * MoveResult
    type PlayerOMoves<'GameState> =
        'GameState -> PlayerOPos -> 'GameState * MoveResult
```

What's wrong with the old design?

So what's wrong with this design? Why was I so unhappy?

First, I was unhappy about the use of the `PlayerXPos` and `PlayerOPos` types. The idea was to wrap a `CellPosition` in a type so that it would be "owned" by a particular player. By doing this, and then having the valid moves be one of these types, I could prevent player X from playing twice, say. That is, after player X has moved, the valid moves for the next run would be wrapped in a `PlayerOPos` type so that only player O could use them.

The problem was that the `PlayerXPos` and `PlayerOPos` types are public, so that a malicious user could have forged one and played twice anyway!

Yes, these types could have been made private by parameterizing them like the game state, but the design would have become very ugly very quickly.

Second, even if the moves *were* made unforgeable, there's that game state floating about.

It's true that the game state internals are private, but a malicious user could have still caused problems by reusing a game state. For example, they could attempt to play one of the valid moves with a game state from a previous turn, or vice versa.

In this particular case, it would not be dangerous, but in general it might be a problem.

So, as you can see, this design was becoming a bit smelly to me, which was why I was becoming unhappy.

What's up with this malicious user?

Why I am assuming that the user of the API will be so malicious -- forging fake moves and all that?

The reason is that I use this as a design guideline. If a malicious user can do something I don't want, then the design is probably not good enough.

In my series on capability based security I point out that by designing for the Principle Of Least Authority ("POLA"), you end up with a good design as a side-effect.

That is, if you design the most minimal interface that the caller needs, then you will both avoid accidental complexity (good design) and increase security (POLA).

I had a little tip in that post: **design for malicious callers and you will probably end up with more modular code**.

I think I will follow my own advice and see where I end up!

Designing for POLA

So, let's design for POLA -- let's give the user the minimal "capability" to do something and no more.

In this case, I want to give the user the capability to mark a specific position with an "X" or "O".

Here's what I had before:

```
type PlayerXMoves =
    GameState * PlayerXPos -> // input
        GameState * MoveResult // output
```

The user is passing in the location (`PlayerXPos`) that they want to play.

But let's now take away the user's ability to choose the position. Why don't I give the user a function, a `MoveCapability` say, that has the position baked in?

```
type MoveCapability =
    GameState -> // input
        GameState * MoveResult // output
```

In fact, why not bake the game state into the function too? That way a malicious user can't pass the wrong game state to me.

This means that there is no "input" at all now -- everything is baked in!

```
type MoveCapability =
    unit -> // no input
        GameState * MoveResult // output
```

But now we have to give the user a whole set of capabilities, one for each possible move they can make. Where do these capabilities come from?

Answer, the `MoveResult` of course! We'll change the `MoveResult` to return a list of capabilities rather than a list of positions.

```
type MoveResult =
    | PlayerXToMove of MoveCapability list
    | PlayerOToMove of MoveCapability list
    | GameWon of Player
    | GameTied
```

Excellent! I'm much happier with this approach.

And now that the `MoveCapability` contains the game state baked in, we don't need the game state to be in the output either!

So our move function has simplified dramatically and now looks like this:

```
type MoveCapability =
    unit -> MoveResult
```

Look ma! No `'GameState` parameter! It's gone!

A quick walkthrough from the UI's point of view

So now let's pretend we are the UI, and let's attempt to use the new design.

- First, assume that we have a list of available capabilities from the previous move.
- Next, the user must pick one of the capabilities (e.g. squares) to play -- they can't just create any old cell position and play it, which is good. But how will the user know which capability corresponds to which square? The capabilities are completely opaque. We can't tell from the outside what they do!
- Then, given that the user has picked a capability somehow, we run it (with no parameters).
- Next we update the display to show the result of the move. But again, how are we going to know what to display? There is no game state to extract the cells from any longer.

Here's some pseudo-code for the UI game loop:

```
// loop while game not over
let rec playMove moveResult =

    let availableCapabilities = // from moveResult

    // get capability from user input somehow
    let capability = ??

    // use the capability
    let newMoveResult = capability()

    // display updated grid
    let cells = ??  // from where

    // play again
    match newMoveResult with
    | PlayerXToMove capabilities ->
        // play another move
        playMove newMoveResult
    | etc
```

Let's deal with the first issue: how does the user know which capability is associated with which square?

The answer is just to create a new structure that "labels" the capability. In this case, with the cell position.

```
type NextMoveInfo = {
    posToPlay : CellPosition
    capability : MoveCapability }
```

And now we must change the `MoveResult` to return a list of these labelled capabilities, rather than the unlabelled ones:

```
type MoveResult =
    | PlayerXToMove of NextMoveInfo list
    | PlayerOToMove of NextMoveInfo list
    | GameWon of Player
    | GameTied
```

Note that the cell position is for the user's information only -- the actual position is still baked into the capability and cannot be forged.

Now for the second issue: how does the UI know what to display as a result of the move? Let's just return that information to it directly in a new structure:

```
/// Everything the UI needs to know to display the board
type DisplayInfo = {
    cells : Cell list
    }
```

And once again, the `MoveResult` must be changed, this time to return the `DisplayInfo` for each case:

```
type MoveResult =
    | PlayerXToMove of DisplayInfo * NextMoveInfo list
    | PlayerOToMove of DisplayInfo * NextMoveInfo list
    | GameWon of DisplayInfo * Player
    | GameTied of DisplayInfo
```

Dealing with circular dependencies

Here's our final design:

```
/// The capability to make a move at a particular location.
/// The gamestate, player and position are already "baked" into the function.
type MoveCapability =
    unit -> MoveResult

/// A capability along with the position the capability is associated with.
/// This allows the UI to show information so that the user
/// can pick a particular capability to exercise.
type NextMoveInfo = {
    // the pos is for UI information only
    // the actual pos is baked into the cap.
    posToPlay : CellPosition
    capability : MoveCapability }

/// The result of a move. It includes:
/// * The information on the current board state.
/// * The capabilities for the next move, if any.
type MoveResult =
    | PlayerXToMove of DisplayInfo * NextMoveInfo list
    | PlayerOToMove of DisplayInfo * NextMoveInfo list
    | GameWon of DisplayInfo * Player
    | GameTied of DisplayInfo
```

But oops! This won't compile!

`MoveCapability` depends on `MoveResult` which depends on `NextMoveInfo` which in turn depends on `MoveCapability` again. But the F# compiler does not allow forward references in general.

Circular dependencies like this are generally frowned upon (I even have a post called "cyclic dependencies are evil"!) and there are normally work-arounds which you can use to remove them.

In this case though, I will link them together using the `and` keyword, which replaces the `type` keyword and is useful for just these kinds of cases.

```
type MoveCapability =
    // etc
and NextMoveInfo = {
    // etc
and MoveResult =
    // etc
```

Revisiting the API

What does the API look like now?

Originally, we had an API with slots for the three use-cases and also a helper function `getCells`:

```
type TicTacToeAPI<'GameState>  =
    {
    newGame : NewGame<'GameState>
    playerXMoves : PlayerXMoves<'GameState>
    playerOMoves : PlayerOMoves<'GameState>
    getCells : GetCells<'GameState>
    }
```

But now, we don't need the `playerXMoves` or `playerOMoves`, because they are returned to us in the `MoveResult` of a previous move.

And `getCells` is no longer needed either, because we are returning the `DisplayInfo` directly now.

So after all these changes, the new API just has a single slot in it and looks like this:

```
type NewGame = unit -> MoveResult

type TicTacToeAPI =
    {
    newGame : NewGame
    }
```

I've changed `NewGame` from a constant to a parameterless function, which is in fact, just a `MoveCapability` in disguise.

The new design in full

Here's the new design in full:

```
module TicTacToeDomain =

    type HorizPosition = Left | HCenter | Right
    type VertPosition = Top | VCenter | Bottom
    type CellPosition = HorizPosition * VertPosition

    type Player = PlayerO | PlayerX

    type CellState =
        | Played of Player
        | Empty

    type Cell = {
        pos : CellPosition
        state : CellState
        }

    /// Everything the UI needs to know to display the board
    type DisplayInfo = {
        cells : Cell list
        }

    /// The capability to make a move at a particular location.
    /// The gamestate, player and position are already "baked" into the function.
    type MoveCapability =
        unit -> MoveResult

    /// A capability along with the position the capability is associated with.
    /// This allows the UI to show information so that the user
    /// can pick a particular capability to exercise.
    and NextMoveInfo = {
        // the pos is for UI information only
        // the actual pos is baked into the cap.
        posToPlay : CellPosition
        capability : MoveCapability }

    /// The result of a move. It includes:
    /// * The information on the current board state.
    /// * The capabilities for the next move, if any.
    and MoveResult =
        | PlayerXToMove of DisplayInfo * NextMoveInfo list
        | PlayerOToMove of DisplayInfo * NextMoveInfo list
        | GameWon of DisplayInfo * Player
        | GameTied of DisplayInfo

    // Only the newGame function is exported from the implementation
    // all other functions come from the results of the previous move
    type TicTacToeAPI  =
        {
        newGame : MoveCapability
        }
```

I'm much happier with this design than with the previous one:

- There is no game state for the UI to worry about.
- There are no type parameters to make it look ugly.
- The api is even more encapsulated -- a malicious UI can do very little now.
- It's shorter -- always a good sign!

The complete application

I have updated the implementation and console application to use this new design.

The complete application is available on GitHub in this gist if you want to play with it.

Surprisingly, the implementation also has become slightly simpler, because all the state is now hidden and there is no need to deal with types like `PlayerXPos` any more.

Logging revisited

In the previous post, I demonstrated how logging could be injected into the API.

But in this design, the capabilities are opaque and have no parameters, so how are we supposed to log that a particular player chose a particular location?

Well, we can't log the capabilities, but we *can* log their context, which we have via the `NextMoveInfo`. Let's see how this works in practice.

First, given a `MoveCapability`, we want to transform it into another `MoveCapability` that also logs the player and cell position used.

Here's the code for that:

```
/// Transform a MoveCapability into a logged version
let transformCapability transformMR player cellPos (cap:MoveCapability) :MoveCapabilit
y =

    // create a new capability that logs the player & cellPos when run
    let newCap() =
        printfn "LOGINFO: %A played %A" player cellPos
        let moveResult = cap()
        transformMR moveResult
    newCap
```

This code works as follows:

- Create a new capability `newCap` function that is parameterless and returns a

  `MoveResult` just like the original one.
- When it is called, log the player and cell position. These are not available from the `MoveCapability` that was passed in, so we have to pass them in explicitly.
- Next, call the original capability and get the result.
- The result itself contains the capabilities for the next move, so we need to recursively transform each capability in the `MoveResult` and return a new `MoveResult`. This is done by the `transformMR` function that is passed in.

Now that we can transform a `MoveCapability`, we can go up a level and transform a `NextMoveInfo`.

```
/// Transform a NextMove into a logged version
let transformNextMove transformMR player (move:NextMoveInfo) :NextMoveInfo =
    let cellPos = move.posToPlay
    let cap = move.capability
    {move with capability = transformCapability transformMR player cellPos cap}
```

This code works as follows:

- Given a `NextMoveInfo`, replace its capability with a transformed one. The output of `transformNextMove` is a new `NextMoveInfo`.
- The cellPos comes from the original move.
- The player and `transformMR` function are not available from the move, so must be passed in explicitly again.

Finally, we need to implement the function that will transform a `MoveResult`:

```
/// Transform a MoveResult into a logged version
let rec transformMoveResult (moveResult:MoveResult) :MoveResult =

    let tmr = transformMoveResult // abbreviate!

    match moveResult with
    | PlayerXToMove (display,nextMoves) ->
        let nextMoves' = nextMoves |> List.map (transformNextMove tmr PlayerX)
        PlayerXToMove (display,nextMoves')
    | PlayerOToMove (display,nextMoves) ->
        let nextMoves' = nextMoves |> List.map (transformNextMove tmr PlayerO)
        PlayerOToMove (display,nextMoves')
    | GameWon (display,player) ->
        printfn "LOGINFO: Game won by %A" player
        moveResult
    | GameTied display ->
        printfn "LOGINFO: Game tied"
        moveResult
```

This code works as follows:

- Given a `MoveResult`, handle each case. The output is a new `MoveResult`.
- For the `GameWon` and `GameTied` cases, log the result and return the original moveResult.
- For the `PlayerXToMove` case, take each of the `NextMoveInfo`s and transform them, passing in the required player (`PlayerX`) and `transformMR` function. Note that the `transformMR` function is a reference to this very function! This means that `transformMoveResult` must be marked with `rec` to allow this self-reference.
- For the `PlayerOToMove` case, do the same as the `PlayerXToMove` case, except change the player to `PlayerO`.

Finally, we can inject logging into the API as a whole by transforming the `MoveResult` returned by `newGame`:

```
/// inject logging into the API
let injectLogging api =

    // create a new API with the functions
    // replaced with logged versions
    { api with
        newGame = fun () -> api.newGame() |> transformMoveResult
        }
```

So there you go. Logging is a bit trickier than before, but still possible.

A warning on recursion

In this code, I've been passing around functions that call each other recursively. When you do this, you have to be careful that you don't unwittingly cause a stack overflow.

In a game like this, when the number of nested calls is guaranteed to be small, then there is no issue. But if you are doing tens of thousands of nested calls, then you should worry about potential problems.

In some cases, the F# compiler will do tail-call optimization, but I suggest that you stress test your code to be sure!

Data-centric vs capability-centric designs

There is an interesting difference between the original design and the new design.

The original design was *data-centric*. Yes, we gave each player a function to use, but it was the *same* function used over and over, with different data passed in each time.

The new design is *function-centric* (or as I prefer, *capability-centric*). There is very little data now. Instead, the result of each function call is *another* set of functions than can be used for the next step, and so on, ad infinitum.

In fact, it reminds me somewhat of a continuation-based approach, except that rather than passing in a continuation, the function itself returns a list of continuations, and then you pick one to use.

Capabilities and RESTful designs -- a match made in heaven

If for some crazy reason you wanted to turn this design into a web service, how would you go about doing that?

In a *data-centric* design, we have a function to call (an endpoint URI in the web API) and then we pass data to it (as JSON or XML). The result of the call is more data that we use to update the display (e.g. the DOM).

But in a *capability-centric* design, where's the data? And how do we pass functions around? It seems like this approach would not work for web services at all.

It might surprise you to know that there *is* a way to do this, and what's more it is exactly the same approach used by a RESTful design using HATEOAS.

What happens is that each capability is mapped to a URI by the server, and then visiting that URI is the same as exercising that capability (e.g. calling the function).

For example, in a web-based application based on this Tic-Tac-Toe design, the server would initially return nine URIs, one for each square. Then, when one of those squares was clicked, and the associated URI visited, the server would return eight new URIs, one for each remaining unplayed square. The URI for the just-played square would not be in this list, which means that it could not be clicked on again.

And of course when you click on one of the eight unplayed squares, the server would now return seven new URIs, and so on.

This model is exactly what REST is supposed to be; you decide what to do next based on the contents of the returned page rather than hard-code the endpoints into your app.

One possible downside of this approach is that it is does not appear to be stateless.

- In the data-centric version, all the data needed for a move was passed in each time, which means that scaling the backend services would be trivial.
- In capability-centric approach though, the state has to be stored somewhere. If the complete game state can be encoded into the URI, then this approach will allow stateless servers as well, but otherwise, some sort of state-storage will be needed.

Some web frameworks have made this function-centred approach a key part of their design, most notably Seaside.

The excellent WebSharper framework for F# also uses something similar, I think (I don't know WebSharper as well as I want to, alas, so correct me if I'm wrong).

Summary

In this post, I tore up my original design and replaced it with an even more function-centric one, which I like better.

But of course it still has all those qualities we love: separation of concerns, an API, a watertight security model, self-documenting code, and logging.

I'm going to stop with Tic-Tac-Toe now -- I think I've wrung it dry! I hope you found these two walkthoughs interesting; I learned a lot myself.

NOTE: The code for this post is available on GitHub in this gist.

Ten reasons not to use a statically typed functional programming language

Are you fed up with all the hype about functional programming? Me too! I thought I'd rant about some reasons why sensible people like us should stay away from it.

Just to be clear, when I say "statically typed functional programming language", I mean languages that also include things such as type inference, immutability by default, and so on. In practice, this means Haskell and the ML-family (including OCaml and F#).

Reason 1: I don't want to follow the latest fad

Like most programmers, I'm naturally conservative and I dislike learning new things. That's why I picked a career in IT.

I don't jump on the latest bandwagon just because all the "cool kids" are doing it -- I wait until things have matured and I can get some perspective.

To me, functional programming just hasn't been around long enough to convince me that it is here to stay.

Yes, I suppose some pedants will claim that ML) and Haskell) have been around almost as long as old favorites like Java and PHP, but I only heard of Haskell recently, so that argument doesn't wash with me.

And look at the baby of the bunch, F#. It's only seven years old, for Pete's sake! Sure, that may be a long time to a geologist, but in internet time, seven years is just the blink of an eye.

So, all told, I would definitely take the cautious approach and wait a few decades to see if this functional programming thing sticks around or whether it is just a flash in the pan.

Reason 2: I get paid by the line

I don't know about you, but the more lines of code I write, the more productive I feel. If I can churn out 500 lines of code in a day, that's a job well done. My commits are big, and my boss can see that I've been busy.

But when I compare code written in a functional language with a good old C-like language, there's so much less code that it scares me.

I mean, just look at this code written in a familiar language:

```
public static class SumOfSquaresHelper
{
   public static int Square(int i)
   {
      return i * i;
   }

   public static int SumOfSquares(int n)
   {
      int sum = 0;
      for (int i = 1; i <= n; i++)
      {
         sum += Square(i);
      }
      return sum;
   }
}
```

and compare it with this:

```
let square x = x * x
let sumOfSquares n = [1..n] |> List.map square |> List.sum
```

That's 17 lines vs. only 2 lines. Imagine that difference multiplied over a whole project!

If I did use this approach, my productivity would drop drastically. I'm sorry -- I just can't afford it.

Reason 3: I love me some curly braces

And that's another thing. What's up with all these languages that get rid of curly braces. How can they call themselves real programming languages?

I'll show you what I mean. Here's a code sample with familiar curly braces.

```
public class Squarer
{
    public int Square(int input)
    {
        var result = input * input;
        return result;
    }

    public void PrintSquare(int input)
    {
        var result = this.Square(input);
        Console.WriteLine("Input={0}. Result={1}", input, result);
    }
}
```

And here's some similar code, but without curly braces.

```
type Squarer() =

    let Square input =
        let result = input * input
        result

    let PrintSquare input =
        let result = Square input
        printf "Input=%i. Result=%i" input result
```

Look at the difference! I don't know about you, but I find the second example a bit disturbing, as if something important is missing.

To be honest, I feel a bit lost without the guidance that curly braces give me.

Reason 4: I like to see explicit types

Proponents of functional languages claim that type inference makes the code cleaner because you don't have to clutter your code with type declarations all the time.

Well, as it happens, I *like* to see type declarations. I feel uncomfortable if I don't know the exact type of every parameter. That's why Java is my favorite language.

Here's a function signature for some ML-ish code. There are no type declarations needed and all types are inferred automatically.

```
let GroupBy source keySelector =
    ...
```

And here's the function signature for similar code in C#, with explicit type declarations.

```
public IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
    IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector
    )
    ...
```

I may be in the minority here, but I like the second version much better. It's important to me to know that the return is of type `IEnumerable<IGrouping<TKey, TSource>>`.

Sure, the compiler will type check this for you and warn you if there is a type mismatch. But why let the compiler do the work when your brain can do it instead?

Ok, I admit that if you do use generics, and lambdas, and functions that return functions, and all the other newfangled stuff, then yes, your type declarations can get really hairy and complex. And it gets really hard to type them properly.

But I have an easy fix for that -- don't use generics and don't pass around functions. Your signatures will be much simpler.

Reason 5: I like to fix bugs

To me, there's nothing quite like the thrill of the hunt -- finding and killing a nasty bug. And if the bug is in a production system, even better, because I'll be a hero as well.

But I've read that in statically typed functional languages, it is much harder to introduce bugs.

That's a bummer.

Reason 6: I live in the debugger

And talking of bug fixing, I spend most of my day in the debugger, stepping through code. Yes, I know I should be using unit tests, but easier said than done, OK?

Anyway, apparently with these statically typed functional languages, if your code compiles, it usually works.

I'm told that you do have to spend a lot of time up front getting the types to match up, but once that is done and it compiles successfully, there is nothing to debug. Where's the fun in that?

Which brings me to...

Reason 7: I don't want to think about every little detail

All this matching up types and making sure everything is perfect sounds tiring to me.

In fact, I hear that you are forced to think about all the possible edge cases, and all the possible error conditions, and every other thing that could go wrong. And you have to do this at the beginning -- you can't be lazy and postpone it till later.

I'd much rather get everything (mostly) working for the happy path, and then fix bugs as they come up.

Reason 8: I like to check for nulls

I'm very conscientious about checking for nulls on every method. It gives me great satisfaction to know that my code is completely bulletproof as a result.

```
void someMethod(SomeClass x)
{
    if (x == null) { throw new NullArgumentException(); }

    x.doSomething();
}
```

Haha! Just kidding! Of course I can't be bothered to put null-checking code everywhere. I'd never get any real work done.

But I've only ever had to deal with one bad crash caused by a NPE. And the business didn't lose too much money during the few weeks I spent looking for the problem. So I'm not sure why this is such a big deal.

Reason 9: I like to use design patterns everywhere

I first read about design patterns in the Design Patterns book (for some reason it's referred to as the Gang of Four book, but I'm not sure why), and since then I have been diligent in using them at all times for all problems. It certainly makes my code look serious and "enterprise-y", and it impresses my boss.

But I don't see any mention of patterns in functional design. How can you get useful stuff done without Strategy, AbstractFactory, Decorator, Proxy, and so on?

Perhaps the functional programmers are not aware of them?

Reason 10: It's too mathematical

Here's some more code for calculating the sum of squares. This is *way* too hard to understand because of all the weird symbols in it.

```
ss=: +/ @: *:
```

Oops, sorry! My mistake. That was J code).

But I do hear that functional programs use strange symbols like `<*>` and `>>=` and obscure concepts called "monads" and "functors".

I don't know why the functional people couldn't stick with things I already know -- obvious symbols like `++` and `!=` and easy concepts such as "inheritance" and "polymorphism".

Summary: I don't get it

You know what. I don't get it. I don't get why functional programming is useful.

What I'd really like is for someone to just show me some real benefits on a single page, instead of giving me too much information.

UPDATE: So now I've read the "everything you need to know on one page" page. But it's too short and simplistic for me.

I'm really looking for something with a bit more depth -- something I can get my teeth into.

And no, don't say that I should read tutorials, and play with examples, and write my own code. I just want to grok it without doing all of that work.

I don't want to have to change the way I think just to learn a new paradigm.

Why I won't be writing a monad tutorial

"A ?newbie', in Haskell, is someone who hasn't yet implemented a compiler. They've only written a monad tutorial" - Pseudonymn

Let's start with a story...

Alice learns to count

Young Alice and her father (who happens to be a mathematician) are visiting a petting zoo...

Alice: Look at those kitties.



Daddy: Aren't they cute. There are *two* of them.

Alice: Look at those doggies.



Daddy: That's right. Can you count? There are *two* doggies.

Alice: Look at those horsies.



Daddy: Yes darling. Do you know what the kitties and doggies and horsies all have in common?

Alice: No. Nothing in common!

Daddy: Well, actually they *do* have something in common. Can you see what it is?

Alice: No! A doggy is not a kitty. A horsie is not a kitty.

Daddy: How about I explain for you? First, let us consider a set S which is strictly well-ordered with respect to set membership and where every element of S is also a subset of S. Does that give you a clue?

Alice: [Bursts into tears]

How not to win friends and influence people

No (sensible) parent would ever try to explain how to count by starting with a formal definition of ordinal numbers.

So why is it that many people feel compelled to explain a concept like monads by emphasizing their formal definition?

That might be fine for a college-level math class, but it plainly does not work for regular programmers, who just want to create something useful.

As an unfortunate result of this approach, though, there is now a whole mystique around the concept of monads. It has become a bridge you must cross on the way to true enlightenment. And there are, of course, a plethora of monad tutorials to help you cross it.

Here's the truth: You *don't* need to understand monads to write useful functional code. This is especially true for F# compared to say, Haskell.

Monads are not a golden hammer. They won't make you any more productive. They won't make your code less buggy.

So really, don't worry about them.

Why I won't be writing a monad tutorial

So this is why I won't be writing a monad tutorial. I don't think it will help people learn about functional programming. If anything, it just creates confusion and anxiety.

Yes, I will use examples of monads in many different posts, but, other than right here, I will try to avoid using the word "monad" anywhere on this site. In fact, it has pride of place on my list of banned words!

Why you should write a monad tutorial

On the other hand, I do think that *you* should write a monad tutorial. When you try to explain something to somebody else, you end up understanding it better yourself.

Here's the process I think you should follow:

1. First, write lots of practical code involving lists, sequences, options, async workflows, computation expressions, etc.
2. As you become more experienced, you will start to use more abstractions, focusing on the shapes of things rather than the details.
3. At some point, you will have an aha! moment -- a sudden insight that all the abstractions have something in common.
4. Bingo! Time to write your monad tutorial!

The key point is that *you have to do it in this order* -- you cannot jump straight to the last step and then work backwards. It is the very act of working your way through the details that enables you to understand the abstraction when you see it.

Good luck with your tutorial -- I'm off to eat a burrito.

Is your programming language unreasonable?

As should be obvious, one of the goals of this site is to persuade people to take F# seriously as a general purpose development language.

But as functional idioms have become more mainstream, and C# has added functional capabilities such as lambdas and LINQ, it seems like C# is "catching up" with F# more and more.

So, ironically, I've now started to hear people say things like this:

- "C# already has most of the features of F#, so why should I bother to switch?"*
- "There is no need to change. All we have do is wait a couple of years and C# will get many of the F# features that provide the most benefits."
- "F# is slightly better than C#, but not so much that it's really worth the effort to move towards it."
- "F# seems really nice, even if it's a bit intimidating. But I can't see a practical purpose to use it over C#."

No doubt, the same comments are being made in the JVM ecosystem about Scala and Clojure vs. Java, now that Java has lambdas too.

So for this post, I'm going to stray away from F#, and focus on C# (and by proxy, other mainstream languages), and try to demonstrate that, even with all the functional features in the world, programming in C# will never be the same as programming in F#.

Before I start, I want to make it clear that I am *not* hating on C#. As it happens I like C# very much; it is one of my favorite mainstream languages, and it has evolved to be very powerful while being consistent and backwards compatible, which is a hard thing to pull off.

But C# is not perfect. Like most mainstream OO languages, it contains some design decisions which no amount of LINQ or lambda goodness can compensate for.

In this post, I'll show you some of the issues that these design decisions cause, and suggest some ways to improve the language to avoid them.

(I'm now going to don my flameproof suit. I think I might need it!)

UPDATE: Many people have seriously misread this post, it seems. So let me be clear:

- I am *not* saying that statically typed languages are "better" than dynamic languages.
- I am *not* saying that FP languages are "better" than OO languages.
- I am *not* saying that being able to reason about code is the most important aspect of a language.

What I *am* saying is:

- Not being able to reason about code has costs that many developers might not be aware of.
- Therefore, being "reasonable" should be one of the (many) factors under consideration when choosing a programming language, not just ignored due to lack of awareness.
- *IF* you want to be able to reason about your code, *THEN* it will be much easier if your language supports the features that I mention.
- The fundamental paradigm of OO (object-identity, behavior-based) is not compatible with "reasonability", and so it will be hard to retrofit existing OO languages to add this quality.

That's it. Thank you!

What is a "reasonable" programming language, anyway?

If you hang around functional programmers, you will often hear the phrase "reason about", as in "we want to reason about our programs".

What does that mean? Why use the word "reason" rather than just "understand"?

The use of "reasoning" goes back to mathematics and logic, but I'm going to use a simple and pragmatic definition:

- "reasoning about the code" means that you can draw conclusions using only the information that you have *right in front of you*, rather than having to delve into other parts of the codebase.

In other words, you can predict the behavior of some code just by looking at it. You may need to understand the interfaces to other components, but you shouldn't need to look inside them to see what they do.

Since, as developers, we spend most of our time looking at code, this is a pretty important aspect of programming!

Of course, there is a huge amount of advice out there on how to do just this: naming guidelines, formatting rules, design patterns, etc., etc.

But can your programming language *by itself* help your code to be more reasonable, more predictable? I think the answer is yes, but I'll let you judge for yourself.

Below, I'll present a series of code fragments. After each snippet, I'm going to ask you what you think the code does. I've deliberately not shown my own comments so that you can think about it and do your own reasoning. After you have thought about it, scroll down to read my opinion.

Example 1

Let's start off by looking at the following code.

- We start with a variable `x` that is assigned the integer `2`.
- Then `DoSomething` is called with `x` as a parameter.
- Then `y` is assigned to `x - 1`.

The question I would ask you is simple: What is the value of `y`?

```
var x = 2;
DoSomething(x);

// What value is y?
var y = x - 1;
```

(scroll down for answer)

The answer is `-1`. Did you get that answer? No? If you can't figure it out, scroll down again.

Trick question! This code is actually JavaScript!

Here's the whole thing:

```
function DoSomething (foo) { x = false}

var x = 2;
DoSomething(x);
var y = x - 1;
```

Yes, it's horrible! `DoSomething` accesses `x` directly rather than through the parameter, and then turns it into a boolean of all things! Then, subtracting 1 from `x` casts it from `false` to `0`, so that `y` is `-1`.

Don't you totally hate this? Sorry to mislead you about the language, but I just wanted to demonstrate how annoying it is when the language behaves in unpredictable ways.

JavaScript is a very useful and important language. But no one would claim that reasonableness was one of its strengths. In fact, most dynamically-typed languages have quirks that make them hard to reason about in this way.

Thanks to static typing and sensible scoping rules, this kind of thing could never happen in C# (unless you tried really hard!) In C#, if you don't match up the types properly, you get a *compile-time* error rather than a *run-time* error.

In other words, C# is much more predictable than JavaScript. Score one for static typing!

So now we have our first requirement for making a language predictable:

How to make your language predictable:

1. Variables should not be allowed to change their type.

C# is looking good compared to JavaScript. But we're not done yet...

UPDATE: This is an admittedly silly example. In retrospect, I could have picked a better one. Yes, I know that no one sensible would ever do this. The point still stands: the JavaScript language does not prevent you from doing stupid things with implicit typecasts.

Example 2

In this next example, we're going to create two instances of the same `Customer` class, with exactly the same data in them.

The question is: Are they equal?

```
// create two customers
var cust1 = new Customer(99, "J Smith");
var cust2 = new Customer(99, "J Smith");

// true or false?
cust1.Equals(cust2);
```

(scroll down for answer)

```
// true or false?
cust1.Equals(cust2);
```

Who knows? It depends on how the `Customer` class has been implemented. This code is *not* predictable.

You'll have to look at whether the class implements `IEquatable` at least, and you'll probably have to look at the internals of the class as well to see exactly what is going on.

But why is this even an issue?

Let me ask you this:

- How often would you NOT want the instances to be equal?
- How often have you had to override the `Equals` method?
- How often have you had a bug caused by *forgetting* to override the `Equals` method?
- How often have you had a bug caused by mis-implementing `GetHashCode` (such as forgetting to change it when the fields that you compare on change)?

Why not make the objects equal by default, and make reference equality testing the special case?

So let's add another item to our list.

How to make your language predictable:

1. Variables should not be allowed to change their type.
2. **Objects containing the same values should be equal by default.**

Example 3

In this next example, I've got two objects containing exactly the same data, but which are instances of different classes.

The question again is: Are they equal?

```
// create a customer and an order
var cust = new Customer(99, "J Smith");
var order = new Order(99, "J Smith");

// true or false?
cust.Equals(order);
```

(scroll down for answer)

```
// true or false?
cust.Equals(order);
```

Who cares! This is almost certainly a bug! Why are you even comparing two different classes like this in the first place?

Compare their names or ids, certainly, but not the objects themselves. This should be a compiler error.

If it isn't, why not? You probably just used the wrong variable name by mistake but now you have a subtle bug in your code. Why does your language let you do this?

So let's add another item to our list.

How to make your language predictable:

1. Variables should not be allowed to change their type.
2. Objects containing the same values should be equal by default.
3. **Comparing objects of different types is a compile-time error.**

UPDATE: Many people have pointed out that you need this when comparing classes related by inheritance. This is true, of course. But what is the cost of this feature? You get the ability to compare subclasses, but you lose the ability to detect accidental errors.

Which is more important in practice? That's for you to decide, I just wanted to make it clear that there are costs associated with the status quo, not just benefits.

Example 4

In this snippet, we're just going to create a `Customer` instance. That's all. Can't get much more basic than that.

```
// create a customer
var cust = new Customer();

// what is the expected output?
Console.WriteLine(cust.Address.Country);
```

Now the question is: what is the expected output of `WriteLine`?

(scroll down for answer)

```
// what is the expected output?
Console.WriteLine(cust.Address.Country);
```

Who knows?

It depends on whether the `Address` property is null or not. And that is something you can't tell without looking at the internals of the `Customer` class again.

Yes, we know that it is a best practice that constructors should initialize all fields at construction time, but why doesn't the language enforce it?

If the address is required, then make it be required in the constructor. And if the address is *not* always required, then make it clear that the `Address` property is optional and might be missing.

So let's add another item to our list of improvements.

How to make your language predictable:

1. Variables should not be allowed to change their type.
2. Objects containing the same values should be equal by default.
3. Comparing objects of different types is a compile-time error.

4. **Objects must *always* be initialized to a valid state. Not doing so is a compile-time error.**

Example 5

In this next example, we're going to:

- Create a customer.
- Add it to a set that uses hashing.
- Do something with the customer object.
- See if the customer is still in the set.

What could possibly go wrong?

```
// create a customer
var cust = new Customer(99, "J Smith");

// add it to a set
var processedCustomers = new HashSet<Customer>();
processedCustomers.Add(cust);

// process it
ProcessCustomer(cust);

// Does the set contain the customer? true or false?
processedCustomers.Contains(cust);
```

So, does the set still contain the customer at the end of this code?

(scroll down for answer)

```
// Does the set contain the customer?
processedCustomers.Contains(cust);
```

Maybe. Maybe not.

It depends on two things:

- First, does the hash code of the customer depend on a *mutable* field, such as an id.
- Second, does `ProcessCustomer` change this field?

If both are true, then the hash will have been changed, and the customer will not longer *appear* to exist in the set (even though it is still in there somewhere!).

This might well cause subtle performance and memory problems (e.g. if the set is a cache).

How could the language prevent this?

One way would be to say that any field or property used in `GetHashCode` must be immutable, while allowing other properties to be mutable. But that is really impractical.

Better to just make the entire `Customer` class immutable instead!

Now if the `Customer` class was immutable, and `ProcessCustomer` wanted to make changes, it would have to return a *new version* of the customer, and the code would look like this:

```
// create a customer
var cust = new ImmutableCustomer(99, "J Smith");

// add it to a set
var processedCustomers = new HashSet<ImmutableCustomer>();
processedCustomers.Add(cust);

// process it and return the changes
var changedCustomer = ProcessCustomer(cust);

// true or false?
processedCustomers.Contains(cust);
```

Notice that the `ProcessCustomer` line has changed to:

```
var changedCustomer = ProcessCustomer(cust);
```

It's clear that `ProcessCustomer` has changed something just by looking at this code. If `ProcessCustomer` *hadn't* changed anything, it wouldn't have needed to return an object at all.

Going back to the question, it's clear that in this implementation the original version of the customer is guaranteed to still be in the set, no matter what `ProcessCustomer` does.

Of course, that doesn't solve the issue of whether the new one or the old one (or both) should be in the set. But unlike the implementation using the mutable customer, this issue is now staring you in the face and won't go unnoticed accidentally.

So immutability FTW!

So that's another item for our list.

How to make your language predictable:

1. Variables should not be allowed to change their type.
2. Objects containing the same values should be equal by default.
3. Comparing objects of different types is a compile-time error.
4. Objects must *always* be initialized to a valid state. Not doing so is a compile-time error.
5. **Once created, objects and collections *must* be immutable.**

Time for a quick joke about immutability:

> "How many Haskell programmers does it take to change a lightbulb?"
>
> "Haskell programmers don't "change" lightbulbs, they "replace" them. And you must also replace the whole house at the same time."

Almost done now -- just one more!

Example 6

In this final example, we'll try to fetch a customer from a `CustomerRepository`.

```
// create a repository
var repo = new CustomerRepository();

// find a customer by id
var customer = repo.GetById(42);

// what is the expected output?
Console.WriteLine(customer.Id);
```

The question is: after we do `customer = repo.GetById(42)`, what is the value of `customer.Id`?

(scroll down for answer)

```
var customer = repo.GetById(42);

// what is the expected output?
Console.WriteLine(customer.Id);
```

It all depends, of course.

If I look at the method signature of `GetById`, it tells me it always returns a `Customer`. But does it *really*?

What happens if the customer is missing? Does `repo.GetById` return `null`? Does it throw an exception? You can't tell just by looking at the code that we've got.

In particular, `null` is a terrible thing to return. It's a turncoat that pretends to be a `Customer` and can be assigned to `Customer` variables with nary a complaint from the compiler, but when you actually ask it to do something, it blows up in your face with an evil cackle. Unfortunately, I can't tell by looking at this code whether a null is returned or not.

Exceptions are a little better, because at least they are typed and contain information about the context. But it's not apparent from the method signature which exceptions might be thrown. The only way that you can know for sure is by looking at the internal source code (and maybe the documentation, if you're lucky and it is up to date).

But now imagine that your language did not allow `null` and did not allow exceptions. What could you do instead?

The answer is, you would be forced to return a special class that might contain *either* a customer *or* an error, like this:

```
// create a repository
var repo = new CustomerRepository();

// find a customer by id and
// return a CustomerOrError result
var customerOrError = repo.GetById(42);
```

The code that processed this "customerOrError" result would then have to test what kind of result it was, and handle each case separately, like this:

```
// handle both cases
if (customerOrError.IsCustomer)
    Console.WriteLine(customerOrError.Customer.Id);

if (customerOrError.IsError)
    Console.WriteLine(customerOrError.ErrorMessage);
```

This is exactly the approach taken by most functional languages. It does help if the language provides conveniences to make this technique easier, such as sum types, but even without that, this approach is still the only way to go if you want to make it obvious what your code is doing. (You can read more about this technique here.)

So that's the last two items to add to our list, at least for now.

How to make your language predictable:

1. Variables should not be allowed to change their type.
2. Objects containing the same values should be equal by default.
3. Comparing objects of different types is a compile-time error.
4. Objects must *always* be initialized to a valid state. Not doing so is a compile-time error.
5. Once created, objects and collections *must* be immutable.
6. **No nulls allowed.**
7. **Missing data or errors must be made explicit in the function signature.**

I could go on, with snippets demonstrating the misuse of globals, side-effects, casting, and so on. But I think I'll stop here -- you've probably got the idea by now!

Can your programming language do *this*?

I hope that it is obvious that making these additions to a programming language will help to make it more reasonable.

Unfortunately, mainstream OO languages like C# are very unlikely to add these features.

First of all, it would be a major breaking change to all existing code.

Second, many of these changes go deeply against the grain of the object-oriented programming model itself.

For example, in the OO model, object identity is paramount, so *of course* equality by reference is the default.

Also, from an OO point of view, how two objects are compared is entirely up to the objects themselves -- OO is all about polymorphic behavior and the compiler needs to stay out of it! Similarly, how objects are constructed and initialized is again entirely up to the object itself. There are no rules to say what should or should not be allowed.

Finally, it is very hard to add non-nullable reference types to a statically typed OO language without also implementing the initialization constraints in point 4. As Eric Lippert himself has said "Non-nullability is the sort of thing you want baked into a type system from day one, not something you want to retrofit 12 years later".

In contrast, most functional programming languages have these "high-predictability" features as a core part of the language.

For example, in F#, all but one of the items on that list are built into the language:

1. Values are not allowed to change their type. (And this even includes implicit casts from int to float, say).
2. Records with the same internal data *ARE* equal by default.
3. Comparing values of different types *IS* a compile-time error.
4. Values *MUST* be initialized to a valid state. Not doing so is a compile-time error.
5. Once created, values *ARE* immutable by default.
6. Nulls are *NOT* allowed, in general.

Item #7 is not enforced by the compiler, but discriminated unions (sum types) are generally used to return errors rather than using exceptions, so that the function signature indicates exactly what the possible errors are.

It's true that when working with F# there are still many caveats. You *can* have mutable values, you *can* create and throw exceptions, and you may indeed have to deal with nulls that come from non-F# code.

But these things are considered code smells and are unusual, rather than being the general default.

Other languages such as Haskell are even purer (and hence even more reasonable) than F#, but even Haskell programs will not be perfect.

In fact, no language can be reasoned about *perfectly* and still be practical. But still, some languages are certainly more reasonable than others.

I think that one of the reasons why many people have become so enthusiastic about functional-style code (and call it "simple" even though it's full of strange symbols!) is exactly this: immutability, and lack of side effects, and all the other functional principles, act together to enforce this reasonability and predictability, which in turn helps to reduce your cognitive burden so that you need only focus on the code in front of you.

Lambdas aren't the solution

So now it should be clear that this list of proposed improvements has nothing to do with language enhancements such as lambdas or clever functional libraries.

In other words, when I focus on reasonability, **I don't care what my language *will* let me do, I care more about what my language *won't* let me do.** I want a language that stops me doing stupid things by mistake.

That is, if I had to choose between language A that didn't allow nulls, or language B that had higher-kinded types but still allowed objects to be null easily, I would pick language A without hesitation.

Questions

Let me see if I can prempt some questions...

Question: These examples are very contrived! If you code carefully and follow good practices, you can write safe code without these features!

Yes, you can. I'm not claiming you can't. But this post is not about writing safe code, it's about *reasoning* about the code. There is a difference.

And it's not about what you can do if you are careful. It's about what can happen if you are not careful!
That is, does your *programming language* (not your coding guidelines, or tests, or IDE, or development practices) give you support for reasoning about your code?

Question: You're telling me that a language *should* have these features. Isn't that very arrogant of you?

Please read carefully. I am not saying that at all. What I *am* saying is that:

- *IF* you want to be able to reason about your code, *THEN* it will be much easier if your

language supports the features that I mention.

If reasoning about your code is not that important to you, then please do feel free to ignore everything I've said!

Question: Focusing on just one aspect of a programming language is too limiting. Surely other qualities are just as important?

Yes, or course they are. I am not a absolutist on this topic. I think that factors such as comprehensive libraries, good tooling, a welcoming community, and the strength of the ecosystem are very important too.

But the purpose of this post was to address the specific comments I mentioned at the beginning, such as: "C# already has most of the features of F#, so why should I bother to switch?".

Question: Why are you dismissing dynamic languages so quickly?

First, my apologies to JavaScript developers for the dig earlier!

I like dynamic languages a lot, and one of my favorite languages, Smalltalk, is completely unreasonable by the standards I've talked about. Luckily, this post is not trying to persuade you which languages are "best" in general, but rather just discussing one aspect of that choice.

Question: Immutable data structures are slow, and there will be lots of extra allocation going on. Won't this affect performance?

This post is not attempting to address the performance impact (or any other aspect) of these features.

But it is indeed a valid question to ask which should have a higher priority: code quality or performance? That's for you to decide, and it depends on the context.

Personally, I would go for safety and quality first, unless there was a compelling reason not to. Here's a sign I like:



Summary

I said just above that this post is not trying to persuade you to pick a language based on "reasonability" alone. But that's not quite true.

If you have already picked a statically typed, high-level language such as C# or Java, then it's clear that reasonability or something like it was an important criterion in your language decision.

In that case, I hope that the examples in this post might have made you more willing to consider using an even more "reasonable" language on your platform of choice (.NET or JVM).

The argument for staying put -- that your current language will eventually "catch up" -- may be true purely in terms of features, but no amount of future enhancements can really change the core design decisions in an OO language. You'll never get rid of nulls, or mutability, or having to override equality all the time.

What's nice about F#, or Scala/Clojure, is that these functional alternatives don't require you to change your ecosystem, but they do immediately improve your code quality.

In my opinion, it's quite a low risk compared with the cost of business as usual.

(I'll leave the issue of finding skilled people, training, support, etc, for another post. But see this, this, this, and this if you're worried about hiring)

We don't need no stinking UML diagrams

In my talk on functional DDD, I often use this slide (*in context*):



Which is of course is a misquote of this famous scene. Oops, I mean this one.

Ok, I might be exaggerating a bit. Some UML diagrams are useful (I like sequence diagrams for example) and in general, I do think a good picture or diagram can be worth 1000 words.

But I believe that, in many cases, using UML for class diagrams is not necessary.

Instead, a concise language like F# (or OCaml or Haskell) can convey the same meaning in a way that is easier to read, easier to write, and most important, easier to turn into *working code*!

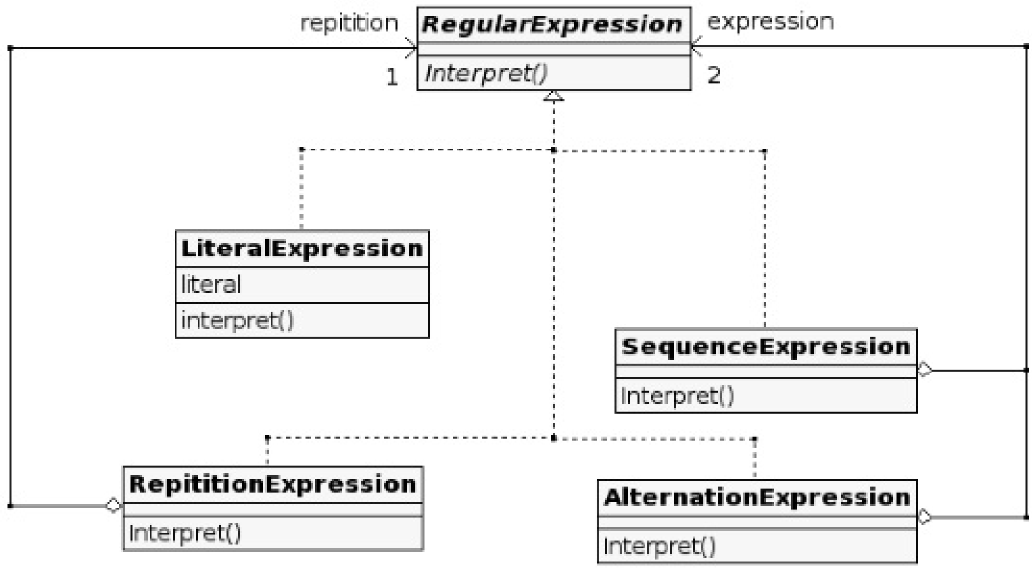
With UML diagrams, you need to translate them to code, with the possibility of losing something in translation. But if the design is documented in your programming language itself, there is no translation phase, and so the design must always be in sync with the implementation.

To demonstrate this in practice, I decided to scour the internet for some good (and not-so-good) UML class diagrams, and convert them into F# code. You can compare them for yourselves.

Regular expressions

Let's start with a classic one: regular expressions (*source*)

Here's the UML diagram:



And here's the F# equivalent:

```
type RegularExpression =
    | Literal of string
    | Sequence of RegularExpression list
    | Alternation of RegularExpression * RegularExpression
    | Repetition of RegularExpression

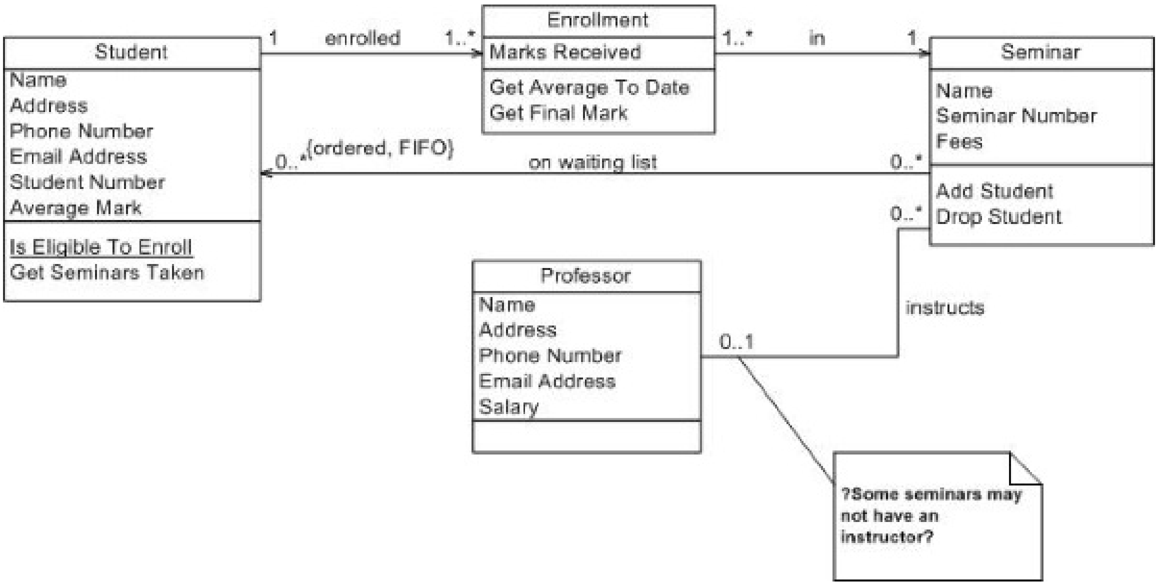
// An interpreter takes a string input and a RegularExpression
// and returns a value of some kind
type Interpret<'a> =  string -> RegularExpression -> 'a
```

That's quite straightforward.

Student enrollment

Here's another classic one: enrollment (*source*).

Here's the UML diagram:



And here's the F# equivalent:

```
type Student = {
    Name: string
    Address: string
    PhoneNumber: string
    EmailAddress: string
    AverageMark: float
    }

type Professor= {
    Name: string
    Address: string
    PhoneNumber: string
    EmailAddress: string
    Salary: int
    }

type Seminar = {
    Name: string
    Number: string
    Fees: float
    TaughtBy: Professor option
    WaitingList: Student list
    }

type Enrollment = {
    Student : Student
    Seminar : Seminar
    Marks: float list
    }

type EnrollmentRepository = Enrollment list

// ==================================
// activities / use-cases / scenarios
// ==================================

type IsElegibleToEnroll = Student -> Seminar -> bool
type GetSeminarsTaken = Student -> EnrollmentRepository -> Seminar list
type AddStudentToWaitingList = Student -> Seminar -> Seminar
```

The F# mirrors the UML diagram, but I find that by writing functions for all the activities rather than drawing pictures, holes in the original requirements are revealed.

For example, in the `GetSeminarsTaken` method in the UML diagram, where is the list of seminars stored? If it is in the `Student` class (as implied by the diagram) then we have a mutual recursion between `Student` and `Seminar` and the whole tree of every student and seminar is interconnected and must be loaded at the same time unless hacks are used.

Instead, for the functional version, I created an `EnrollmentRepository` to decouple the two classes.

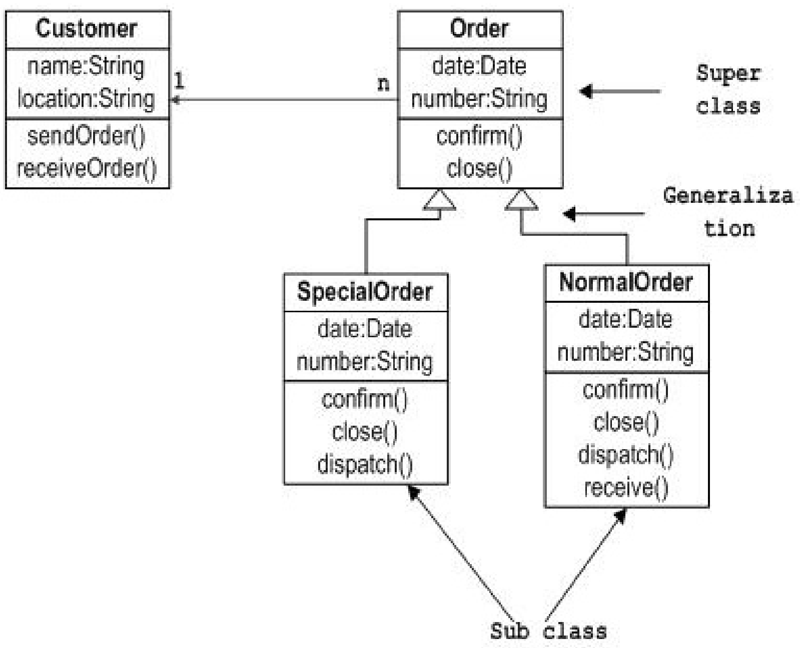
Similarly, it's not clear how enrollment actually works, so I created an `EnrollStudent` function to make it clear what inputs are needed.

```
type EnrollStudent = Student -> Seminar -> Enrollment option
```

Because the function returns an `option`, it is immediately clear that enrollment might fail (e.g student is not eligible to enroll, or is enrolling twice by mistake).

Order and customer

Here's another one (*source*).



And here's the F# equivalent:

```
type Customer = {name:string; location:string}

type NormalOrder = {date: DateTime; number: string; customer: Customer}
type SpecialOrder = {date: DateTime; number: string; customer: Customer}
type Order =
    | Normal of NormalOrder
    | Special of SpecialOrder

// these three operations work on all orders
type Confirm =  Order -> Order
type Close =  Order -> Order
type Dispatch =  Order -> Order

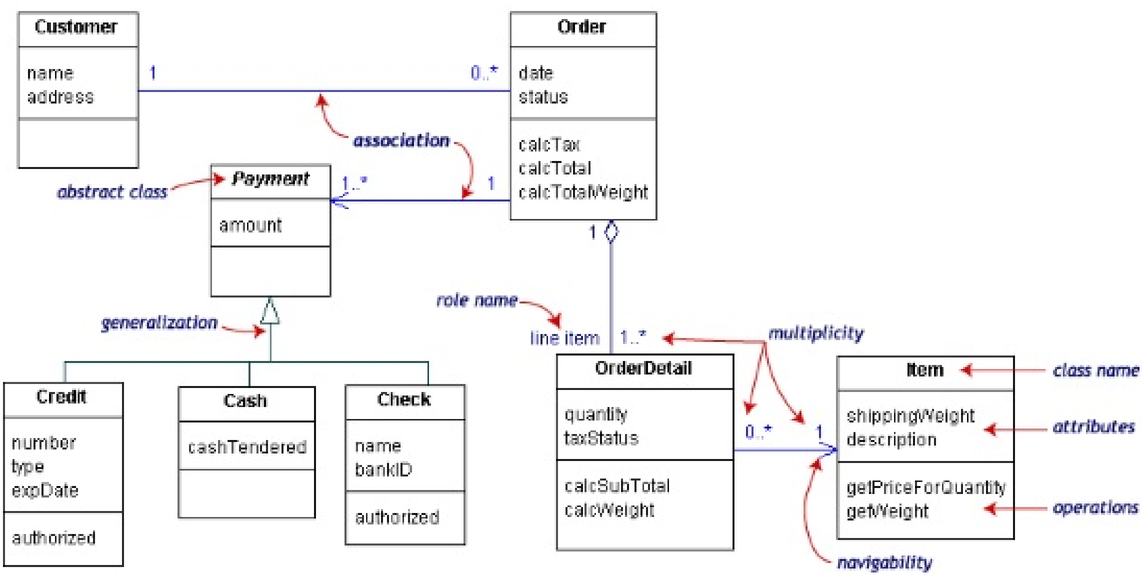
// this operation only works on Special orders
type Receive =  SpecialOrder -> SpecialOrder
```

I'm just copying the UML diagram, but I have to say that I hate this design. It's crying out to have more fine grained states.

In particular, the `Confirm` and `Dispatch` functions are horrible -- they give no idea of what else is needed as input or what the effects will be. This is where writing real code can force you to think a bit more deeply about the requirements.

Order and customer, version 2

Here's a much better version of orders and customers (*source*).



And here's the F# equivalent:

```
type Date = System.DateTime
```

```
// == Customer related ==

type Customer = {
    name:string
    address:string
    }

// == Item related ==

type [<Measure>] grams

type Item = {
    shippingWeight: int<grams>
    description: string
    }

type Qty = int
type Price = decimal


// == Payment related ==

type PaymentMethod =
    | Cash
    | Credit of number:string * cardType:string * expDate:Date
    | Check of name:string * bankID: string

type Payment = {
    amount: decimal
    paymentMethod : PaymentMethod
    }

// == Order related ==

type TaxStatus = Taxable | NonTaxable
type Tax = decimal

type OrderDetail = {
    item: Item
    qty: int
    taxStatus : TaxStatus
    }

type OrderStatus = Open | Completed

type Order = {
    date: DateTime;
    customer: Customer
    status: OrderStatus
    lines: OrderDetail list
    payments: Payment list
    }
```

```
// ==================================
// activities / use-cases / scenarios
// ==================================
type GetPriceForQuantity = Item -> Qty -> Price

type CalcTax = Order -> Tax
type CalcTotal = Order -> Price
type CalcTotalWeight = Order -> int<grams>
```

I've done some minor tweaking, adding units of measure for the weight, creating types to represent `Qty` and `Price`.

Again, this design might be improved with more fine grained states, such as creating a separate `AuthorizedPayment` type (to ensure that an order can only be paid with authorized payments) and a separate `PaidOrder` type (e.g. to stop you paying for the same order twice).

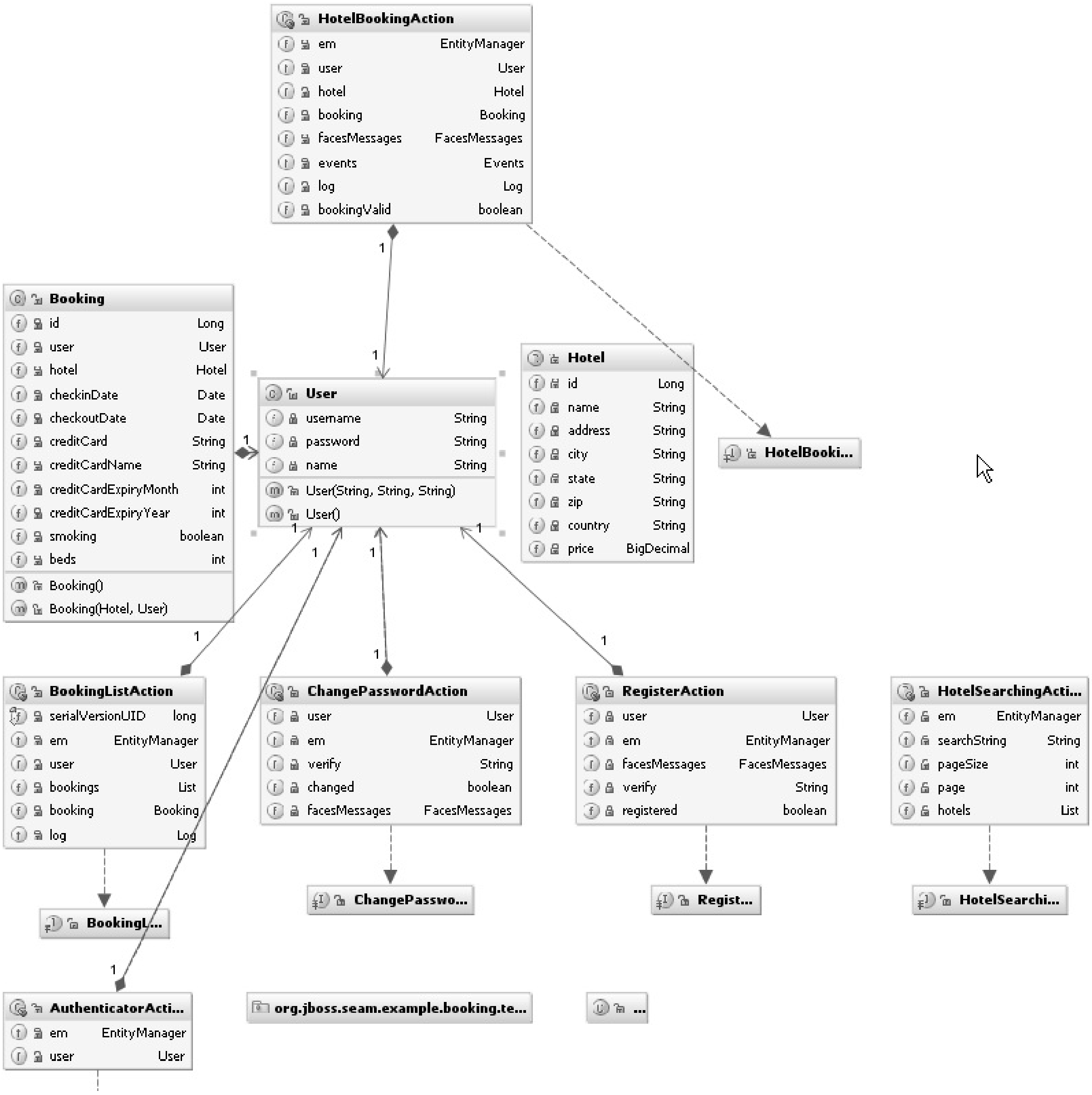
Here's the kind of thing I mean:

```
// Try to authorize a payment. Note that it might fail
type Authorize =  UnauthorizedPayment -> AuthorizedPayment option

// Pay an unpaid order with an authorized payment.
type PayOrder = UnpaidOrder -> AuthorizedPayment -> PaidOrder
```

Hotel Booking

Here's one from the JetBrains IntelliJ documentation (*source*).



Here's the F# equivalent:

```
type Date = System.DateTime

type User = {
    username: string
    password: string
    name: string
    }

type Hotel = {
    id: int
    name: string
    address: string
    city: string
    state: string
    zip: string
    country: string
```

```
    price: decimal
    }

type CreditCardInfo = {
    card: string
    name: string
    expiryMonth: int
    expiryYear: int
    }

type Booking = {
    id: int
    user: User
    hotel: Hotel
    checkinDate: Date
    checkoutDate: Date
    creditCardInfo: CreditCardInfo
    smoking: bool
    beds: int
    }

// What are these?!? And why are they in the domain?
type EntityManager = unit
type FacesMessages = unit
type Events = unit
type Log = unit

type BookingAction = {
    em: EntityManager
    user: User
    hotel: Booking
    booking: Booking
    facesMessages : FacesMessages
    events: Events
    log: Log
    bookingValid: bool
    }

type ChangePasswordAction = {
    user: User
    em: EntityManager
    verify: string
    booking: Booking
    changed: bool
    facesMessages : FacesMessages
    }

type RegisterAction = {
    user: User
    em: EntityManager
    facesMessages : FacesMessages
    verify: string
    registered: bool
```

```
    }
```

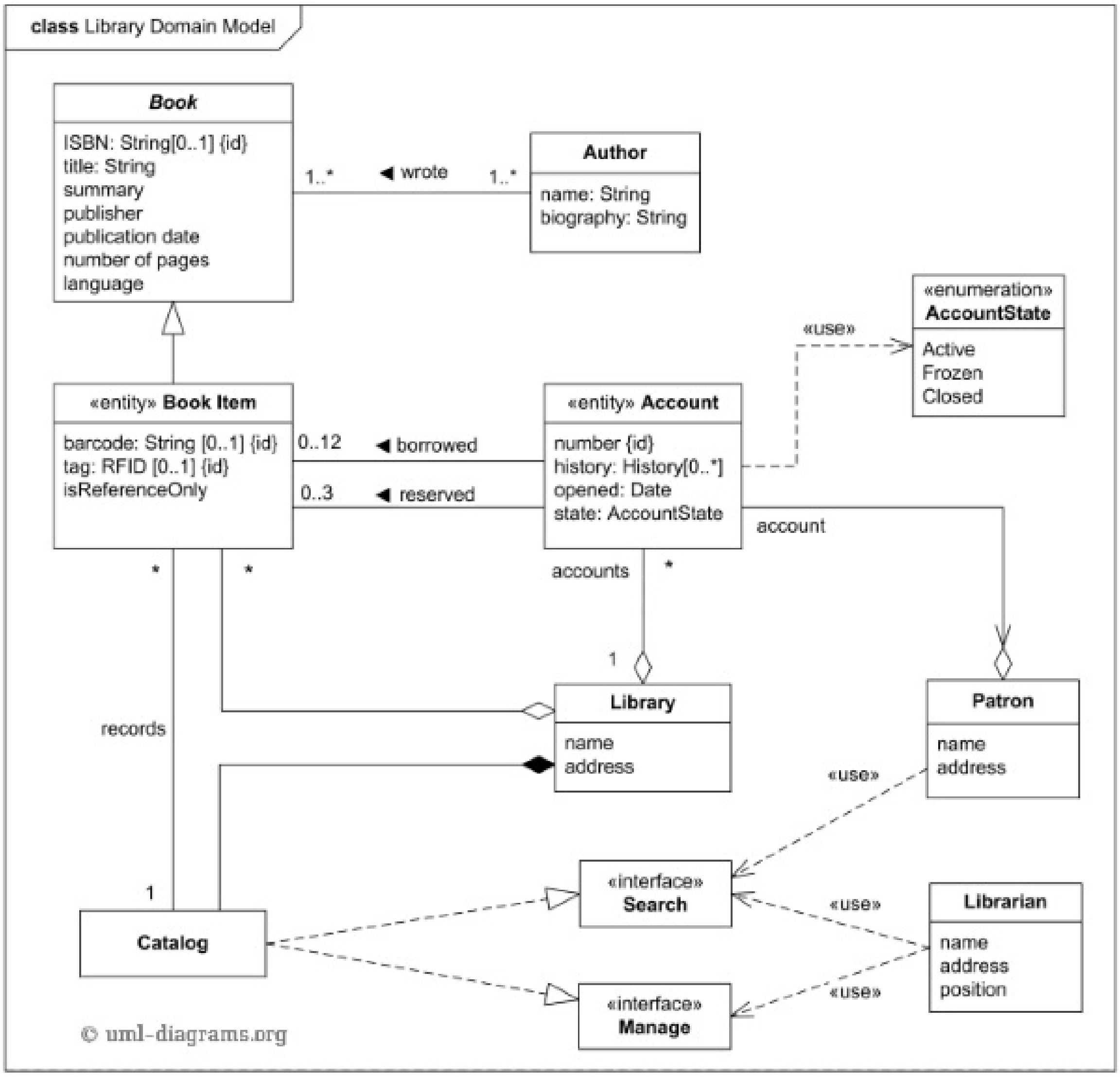
I have to stop there, sorry. The design is driving me crazy. I can't even.

What are these `EntityManager` and `FacesMessages` fields? And logging is important of course, but why is `Log` a field in the domain object?

By the way, in case you think that I am deliberately picking bad examples of UML design, all these diagrams come from the top results in an image search for "uml class diagram".

Library

This one is better, a library domain (*source*).



Here's the F# equivalent. Note that because it is code, I can add comments to specific types and fields, which is doable but awkward with UML.

Note also that I can say `ISBN: string option` to indicate an optional ISBN rather that the awkward `[0..1]` syntax.

```
type Author = {
    name: string
    biography: string
    }

type Book = {
    ISBN: string option
    title: string
    author: Author
    summary: string
    publisher: string
    publicationDate: Date
    numberOfPages: int
    language: string
    }

type Library = {
    name: string
    address: string
    }

// Each physical library item - book, tape cassette, CD, DVD, etc. could have its own
item number.
// To support it, the items may be barcoded. The purpose of barcoding is
// to provide a unique and scannable identifier that links the barcoded physical item
// to the electronic record in the catalog.
// Barcode must be physically attached to the item, and barcode number is entered into

// the corresponding field in the electronic item record.
// Barcodes on library items could be replaced by RFID tags.
// The RFID tag can contain item's identifier, title, material type, etc.
// It is read by an RFID reader, without the need to open a book cover or CD/DVD case
// to scan it with barcode reader.
type BookItem = {
    barcode: string option
    RFID: string option
    book: Book
    /// Library has some rules on what could be borrowed and what is for reference onl
y.
    isReferenceOnly: bool
    belongsTo: Library
    }

type Catalogue = {
    belongsTo: Library
    records : BookItem list
    }

type Patron = {
```

```
    name: string
    address: string
    }

type AccountState = Active | Frozen | Closed

type Account = {
    patron: Patron
    library: Library
    number: int
    opened: Date

    /// Rules are also defined on how many books could be borrowed
    /// by patrons and how many could be reserved.
    history: History list

    state: AccountState
    }

and History = {
    book : BookItem
    account: Account
    borrowedOn: Date
    returnedOn: Date option
    }
```

Since the Search and Manage interfaces are undefined, we can just use placeholders (`unit`) for the inputs and outputs.

```
type Librarian = {
    name: string
    address: string
    position: string
    }

/// Either a patron or a librarian can do a search
type SearchInterfaceOperator =
    | Patron of Patron
    | Librarian of Librarian

type SearchRequest = unit // to do
type SearchResult = unit // to do
type SearchInterface = SearchInterfaceOperator -> Catalogue -> SearchRequest -> Search
Result

type ManageRequest = unit // to do
type ManageResult = unit // to do

/// Only librarians can do management
type ManageInterface = Librarian -> Catalogue -> ManageRequest -> ManageResult
```

Again, this might not be the perfect design. For example, it's not clear that only `Active` accounts could borrow a book, which I might represent in F# as:

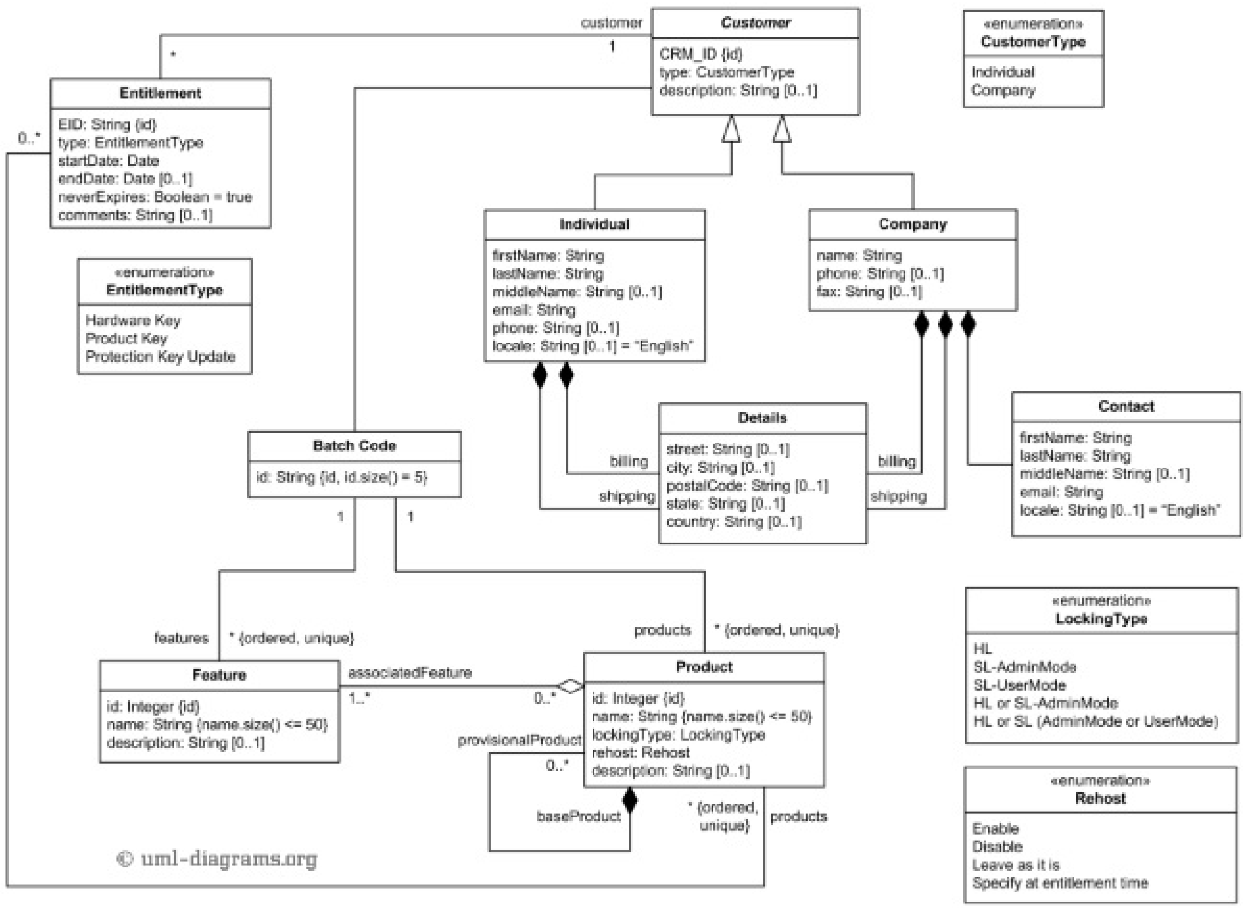
```
type Account =
    | Active of ActiveAccount
    | Closed of ClosedAccount

/// Only ActiveAccounts can borrow
type Borrow = ActiveAccount -> BookItem -> History
```

If you want to see a more modern approach to modelling this domain using CQRS and event sourcing, see this post.

Software licensing

The final example is from a software licensing domain (*source*).



Here's the F# equivalent.

```
open System
type Date = System.DateTime
type String50 = string
```

```
type String5 = string

// ==========================
// Customer-related
// ==========================

type AddressDetails = {
    street : string option
    city : string option
    postalCode : string option
    state : string option
    country : string option
    }

type CustomerIdDescription = {
    CRM_ID : string
    description : string
    }

type IndividualCustomer = {
    idAndDescription : CustomerIdDescription
    firstName : string
    lastName : string
    middleName : string option
    email : string
    phone : string option
    locale : string option // default :  "English"
    billing : AddressDetails
    shipping : AddressDetails
    }

type Contact = {
    firstName : string
    lastName : string
    middleName : string option
    email : string
    locale : string option // default :  "English"
    }

type Company = {
    idAndDescription : CustomerIdDescription
    name : string
    phone : string option
    fax : string option
    contact: Contact
    billing : AddressDetails
    shipping : AddressDetails
    }

type Customer =
    | Individual of IndividualCustomer
    | Company of Company
```

```
// ==========================
// Product-related
// ==========================

/// Flags can be ORed together
[<Flags>]
type LockingType =
    | HL
    | SL_AdminMode
    | SL_UserMode

type Rehost =
    | Enable
    | Disable
    | LeaveAsIs
    | SpecifyAtEntitlementTime

type BatchCode = {
    id : String5
    }

type Feature = {
    id : int
    name : String50
    description : string option
    }

type ProductInfo = {
    id : int
    name : String50
    lockingType : LockingType
    rehost : Rehost
    description : string option
    features: Feature list
    bactchCode: BatchCode
    }

type Product =
    | BaseProduct of ProductInfo
    | ProvisionalProduct of ProductInfo * baseProduct:Product

// ==========================
// Entitlement-related
// ==========================

type EntitlementType =
    | HardwareKey
    | ProductKey
    | ProtectionKeyUpdate

type Entitlement = {
    EID : string
    entitlementType : EntitlementType
```

```
    startDate : Date
    endDate : Date option
    neverExpires: bool
    comments: string option
    customer: Customer
    products: Product list
    }
```

This diagram is just pure data and no methods, so there are no function types. I have a feeling that there are some important business rules that have not been captured.

For example, if you read the comments in the source, you'll see that there are some interesting constraints around `EntitlementType` and `LockingType`. Only certain locking types can be used with certain entitlement types.

That might be something that we could consider modelling in the type system, but I haven't bothered. I've just tried to reproduct the UML as is.

Summary

I think that's enough to get the idea.

My general feeling about UML class diagrams is that they are OK for a sketch, if a bit heavyweight compared to a few lines of code.

For detailed designs, though, they are not nearly detailed enough. Critical things like context and dependencies are not at all obvious. In my opinion, none of the UML diagrams I've shown have been good enough to write code from, even as a basic design.

Even more seriously, a UML diagram can be very misleading to non-developers. It looks "official" and can give the impression that the design has been thought about deeply, when in fact the design is actually shallow and unusable in practice.

Disagree? Let me know in the comments!

Introvert and extrovert programming languages

> What's the difference between an introvert and extrovert mathematician?
> An introvert mathematician looks at his shoes while talking to you; an extrovert mathematician looks at *your* shoes.

For a long time, I've been aware of differences in how programming languages present themselves to the outside world.

If programming languages had personalities, I would be tempted to call some of them "introvert", and some of them "extrovert".

An extrovert programming language is all about the outside world, never happier than when partying with IO and external data sources.

On the other hand, an introvert programming language is quite happy to be alone and would prefer not to deal with the outside world, if possible. Sure, it can be social and work with IO when it needs to, but it finds the activity quite tiring, and is relieved when IO has gone home and it can go back to reading a good book.

What's interesting is that you can tell a lot about the personality type of a language by looking at what is considered important in a user guide or tutorial.

For example, the classic "C Programming Language" book famously has `printf("hello, world\n")` at the very beginning, and most other C books follow the same pattern.

And indeed, C *is* a very extrovert language. Coding examples are littered with files and console IO. Similarly, you can tell that PHP, Python and Perl are equally extrovert with just one glance at their manuals.

In fact, I would say that *all* of the most popular languages are extrovert, and the reasons are obvious. They ooze confidence, they make friends easily, and they get things done.

On the other hand, I would say that Haskell is a great example of an introvert language.

For example, in the book "Learn You A Haskell" the "hello world" example doesn't appear until chapter 9! And in "Real World Haskell", IO is not invited to dinner until chapter 7.

If you don't have the full manual handy, another telling clue that a language is introverted is if it early on introduces you to its close friend, the Fibonacci function. Introvert languages love recursion!

Now, just as in the real world, introvert languages are misunderstood by extroverts. They are accused of being too arrogant, too serious, too risk-averse.

But that's not fair -- introvert languages are really just more reflective and thoughtful, and thus more likely to have deep insights than the shallow, yapping extroverts.

But...

> "All generalisations are false including this one"

You might think that imperative and OO languages would be extrovert, while languages with more declarative paradigms (functional, logic) would be introvert, but that is not always the case.

For example, SQL is a declarative language, but its whole purpose in life is data-processing, which makes it extrovert in my book.

And nearer to home, F# is a functional-first language, but is very happy to do IO, and in fact has excellent support for real-world data processing via type providers and Deedle.

Just as people are not all one or the other, so with programming languages. There is a range. Some languages are extremely extrovert, some extremely introvert, and some in-between.

A lot of people say I am egocentric -- but enough about them

> "Reality is merely an illusion, albeit a very persistent one."

Some languages do not fall so neatly into this personality type spectrum.

Take Smalltalk for example.

In many ways, Smalltalk is *extremely* extrovert. It has lots of support for user interface interaction, and itself was one of the first graphical development environments.

But there's a problem. Sure, it's friendly and chatty and great at intense one-to-one conversations, but it has a dark side -- it doesn't play well with others. It only reluctantly acknowledges the operating system, and rather than dealing with messiness of external libraries, prefers to implement things in its own perfect way.

Most Lisps have the same failing as well. Idealistic and elegant, but rather isolated. I'm going to call this personality type solipsistic.

In their defence, many important programming concepts were first developed in solipsistic languages. But alas, despite their ardent followers, they never gain the widespread recognition they deserve.

Where Do We Come From? What Are We? Where Are We Going?

And that brings me, inevitably, to the all-devouring black hole that is JavaScript. Where does that fit in this silly scheme?

Obviously, its original purpose was to to aid communication with the user (and also, for animating monkeys), so at first glance it would seem to be extrovert.

But the fact that it runs in a sandbox (and until somewhat recently didn't do many kinds of IO) makes me think otherwise. The clincher for me is node.js. Need a server language? Let's create one in our own image! Let's write all the libraries from scratch!

But it seems to have worked, for now at least. So, solipsism for the win!

Concluding Unscientific Postscript

> "Nature has made us frivolous to console us for our miseries"

At this point, I should wrap up with some profound observations about different programming communities and what their language preference reveals about them.

But really, this is a just a bit of whimsy, so I won't.

Swapping type-safety for high performance using compiler directives

TL;DR; An experiment: you can use lots of domain modelling types at development time and swap them out for a more performant implementation later using compiler directives.

Domain Modelling vs. Performance

I am a big fan of using types for domain modelling -- lots and lots and *lots* of types!

These types act both as documentation and as a compile time constraint to ensure that only valid data is used.

For example, if I have two types `CustomerId` and `OrderId` , I can represent them as separate types:

```
type CustomerId = CustomerId of int
type OrderId = OrderId of int
```

and by doing this, I guarantee that I can't use an `OrderId` where I need an `CustomerId` .

The problem is that adding a layer of indirection like this can affect performance:

- the extra indirection can cause data access to be much slower.
- the wrapper class needs extra memory, creating memory pressure.
- this in turn triggers the garbage collector more often, which can often be the cause of performance problems in managed code.

In general, I don't generally worry about micro-performance like this at design-time. Many many things will have a *much* bigger impact on performance, including any kind of I/O, and the algorithms you choose.

As a result, I am very much *against* doing micro-benchmarks out of context. You should always profile a real app in a real context, rather than worrying too much over things that might not be important.

Having said that, I am now going to do some micro-benchmarks!

Micro-benchmarking a wrapper type

Let's see how a wrapper type fares when used in large numbers. Let's say we want to:

- create ten million customer ids
- then, map over them twice
- then, filter them

Admittedly, it's a bit silly adding 1 to a customer id -- we'll look at a better example later.

Anyway, here's the code:

```
// type is an wrapper for a primitive type
type CustomerId = CustomerId of int

// create two silly functions for mapping and filtering
let add1ToCustomerId (CustomerId i) =
    CustomerId (i+1)

let isCustomerIdSmall (CustomerId i) =
    i < 100000

// --------------------------------
// timed with a 1 million element array
// --------------------------------
#time
Array.init 1000000 CustomerId
// map it
|> Array.map add1ToCustomerId
// map it again
|> Array.map add1ToCustomerId
// filter it
|> Array.filter isCustomerIdSmall
|> ignore
#time
```

The code sample above is available on GitHub.

(Again, let me stress that this is a terrible way to profile code!)

A typical timed result looks like this:

```
Real: 00:00:00.296, CPU: 00:00:00.296, GC gen0: 6, gen1: 4, gen2: 0
```

That is, it takes about 0.3 seconds to do those steps, and it creates quite a bit of garbage, triggering four gen1 collections. If you are not sure what "gen0", "gen1", and "gen2" mean, then this is a good place to start.

DISCLAIMER: I'm going to be doing all my benchmarking in F# interactive. Compiled code with optimizations might have a completely different performance profile. Past performance is no guarantee of future results. Draw conclusions at your own risk. Etc., etc.

If we increase the array size to 10 million, we get a more than 10x slower result:

```
Real: 00:00:03.489, CPU: 00:00:03.541, GC gen0: 68, gen1: 46, gen2: 2
```

That is, it takes about 3.5 seconds to do those steps, and it creates *a lot* of garbage, including a few gen2 GC's, which are really bad. In fact, you might even get an "out of memory" exception, in which case, you'll have to restart F# Interactive!

So what are the alternatives to using a wrapper? There are two common approaches:

- Using type aliases
- Using units of measure

Let's start with type aliases.

Using type aliases

In the type alias approach, I would simply dispense with the wrapper, but keep the type around as documentation.

```
type CustomerId = int
type OrderId = int
```

If I want to use the type as documentation, I must then annotate the functions appropriately.

For example, in the `add1ToCustomerId` below both the parameter and the return value have been annotated so that it has the type `CustomerId -> CustomerId` rather than `int -> int`.

```
let add1ToCustomerId (id:CustomerId) :CustomerId =
    id+1
```

Micro-benchmarking a type alias

Let's create another micro-benchmark:

```
type CustomerId = int

// create two silly functions for mapping and filtering
let add1ToCustomerId (id:CustomerId) :CustomerId =
    id+1
// val add1ToCustomerId : id:CustomerId -> CustomerId

let isCustomerIdSmall (id:CustomerId) =
    id < 100000
// val isCustomerIdSmall : id:CustomerId -> bool

// --------------------------------
// timed with a 1 million element array
// --------------------------------
#time
Array.init 1000000 (fun i -> i)
// map it
|> Array.map add1ToCustomerId
// map it again
|> Array.map add1ToCustomerId
// filter it
|> Array.filter isCustomerIdSmall
|> Array.length
#time
```

The code sample above is available on GitHub.

The results are spectacularly better!

```
Real: 00:00:00.017, CPU: 00:00:00.015, GC gen0: 0, gen1: 0, gen2: 0
```

It takes about 17 milliseconds to do those steps, and more importantly, very little garbage was generated.

If we increase the array size to 10 million, we get a 10x slower result, but still no garbage:

```
Real: 00:00:00.166, CPU: 00:00:00.156, GC gen0: 0, gen1: 0, gen2: 0
```

Compared with the earlier version at over three seconds, that's excellent.

Problems with type aliases

Alas, the problem with type aliases is that we have completely lost type safety now!

To demonstrate, here's some code that creates a `CustomerId` and an `OrderId` :

```
type CustomerId = int
type OrderId = int

// create two
let cid : CustomerId = 12
let oid : OrderId = 12
```

And sadly, the two ids compare equal, and we can pass an `OrderId` to function expecting a `CustomerId` without any complaint from the compiler.

```
cid = oid              // true

// pass OrderId to function expecting a CustomerId
add1ToCustomerId oid   // CustomerId = 13
```

Ok, so that doesn't look promising! What next?

Using units of measure

The other common option is to use units of measure to distinguish the two types, like this:

```
type [<Measure>] CustomerIdUOM
type [<Measure>] OrderIdUOM

type CustomerId = int<CustomerIdUOM>
type OrderId = int<OrderIdUOM>
```

`CustomerId` and `OrderId` are still two different types, but the unit of measure is erased, so by the time the JIT sees it the type looks like an primitive int.

We can see that this is true when we time the same steps as before:

```
// create two silly functions for mapping and filtering
let add1ToCustomerId id  =
    id+1<CustomerIdUOM>

let isCustomerIdSmall i =
    i < 100000<CustomerIdUOM>

// --------------------------------
// timed with a 1 million element array
// --------------------------------
#time
Array.init 1000000 (fun i -> LanguagePrimitives.Int32WithMeasure<CustomerIdUOM> i)
// map it
|> Array.map add1ToCustomerId
// map it again
|> Array.map add1ToCustomerId
// filter it
|> Array.filter isCustomerIdSmall
|> ignore
#time
```

The code sample above is available on GitHub.

A typical timed result looks like this:

```
Real: 00:00:00.022, CPU: 00:00:00.031, GC gen0: 0, gen1: 0, gen2: 0
```

Again, the code is very fast (22 milliseconds), and just as importantly, very little garbage was generated again.

If we increase the array size to 10 million, we maintain the high performance (just as with the type alias approach) and still no garbage:

```
Real: 00:00:00.157, CPU: 00:00:00.156, GC gen0: 0, gen1: 0, gen2: 0
```

Problems with units of measure

The advantage of units of measure is that the `CustomerId` and `OrderId` types are incompatible, so we get the type safety that we want.

But I find them unsatisfactory from an esthetic point of view. I like my wrapper types!

And also, units of measure are really meant to be used with numeric values. For example, I can create a customer id and order id:

```
let cid = 12<CustomerIdUOM>
let oid = 4<OrderIdUOM>
```

and then I can divide CustomerId(12) by OrderId(4) to get three...

```
let ratio = cid / oid
// val ratio : int<CustomerIdUOM/OrderIdUOM> = 3
```

Three what though? Three customer ids per order id? What does that even mean?

Yes, surely this will never happen in practice, but still it bothers me!

Using compiler directives to get the best of both worlds

Did I mention that I really like wrapper types? I really like them up until I get a call saying that production systems are having performance hiccups because of too many big GCs.

So, can we get the best of both worlds? Type-safe wrapper types AND fast performance?

I think so, if you are willing to put up with some extra work during development and build.

The trick is to have *both* the "wrapper type" implemention and the "type alias" implementation available to you, and then switch between them based on a compiler directive.

For this to work:

- you will need to tweak your code to not access the type directly, but only via functions and pattern matching.
- you will need to create a "type alias" implementation that implements a "constructor", various "getters" and for pattern matching, active patterns.

Here's an example, using the `COMPILED` and `INTERACTIVE` directives so that you can play with it interactively. Obviously, in real code, you would use your own directive such as `FASTTYPES` or similar.

```
#if COMPILED  // uncomment to use aliased version
//#if INTERACTIVE // uncomment to use wrapped version

// type is an wrapper for a primitive type
type CustomerId = CustomerId of int

// constructor
let createCustomerId i = CustomerId i

// get data
let customerIdValue (CustomerId i) = i

// pattern matching
// not needed

#else
// type is an alias for a primitive type
type CustomerId = int

// constructor
let inline createCustomerId i :CustomerId = i

// get data
let inline customerIdValue (id:CustomerId) = id

// pattern matching
let inline (|CustomerId|) (id:CustomerId) :int = id

#endif
```

You can see that for both versions I've created a constructor `createCustomerId` and a getter `customerIdValue` and, for the type alias version, an active pattern that looks just like `CustomerId` .

With this code in place, we can use `CustomerId` without caring about the implementation:

```
// test the getter
let testGetter c1 c2 =
    let i1 = customerIdValue c1
    let i2 = customerIdValue c2
    printfn "Get inner value from customers %i %i" i1 i2
// Note that the signature is as expected:
// c1:CustomerId -> c2:CustomerId -> unit

// test pattern matching
let testPatternMatching c1 =
    let (CustomerId i) = c1
    printfn "Get inner value from Customers via pattern match: %i" i

    match c1 with
    | CustomerId i2 -> printfn "match/with %i" i
// Note that the signature is as expected:
// c1:CustomerId -> unit

let test() =
    // create two ids
    let c1 = createCustomerId 1
    let c2 = createCustomerId 2
    let custArray : CustomerId [] = [| c1; c2 |]

    // test them
    testGetter c1 c2
    testPatternMatching c1
```

And now we can run the *same* micro-benchmark with both implementations:

```
// create two silly functions for mapping and filtering
let add1ToCustomerId (CustomerId i) =
    createCustomerId (i+1)

let isCustomerIdSmall (CustomerId i) =
    i < 100000

// --------------------------------
// timed with a 1 million element array
// --------------------------------
#time
Array.init 1000000 createCustomerId
// map it
|> Array.map add1ToCustomerId
// map it again
|> Array.map add1ToCustomerId
// filter it
|> Array.filter isCustomerIdSmall
|> Array.length
#time
```

The code sample above is available on GitHub.

The results are similar to the previous examples. The aliased version is much faster and does not create GC pressure:

```
// results using wrapped version
Real: 00:00:00.408, CPU: 00:00:00.405, GC gen0: 7, gen1: 4, gen2: 1

// results using aliased version
Real: 00:00:00.022, CPU: 00:00:00.031, GC gen0: 0, gen1: 0, gen2: 0
```

and for the 10 million element version:

```
// results using wrapped version
Real: 00:00:03.199, CPU: 00:00:03.354, GC gen0: 67, gen1: 45, gen2: 2

// results using aliased version
Real: 00:00:00.239, CPU: 00:00:00.202, GC gen0: 0, gen1: 0, gen2: 0
```

A more complex example

In practice, we might want something more complex than a simple wrapper.

For example, here is an `EmailAddress` (a simple wrapper type, but constrained to be non-empty and containing a "@") and some sort of `Activity` record that stores an email and the number of visits, say.

```
module EmailAddress =
    // type with private constructor
    type EmailAddress = private EmailAddress of string

    // safe constructor
    let create s =
        if System.String.IsNullOrWhiteSpace(s) then
            None
        else if s.Contains("@") then
            Some (EmailAddress s)
        else
            None

    // get data
    let value (EmailAddress s) = s

module ActivityHistory =
    open EmailAddress

    // type with private constructor
    type ActivityHistory = private {
        emailAddress : EmailAddress
        visits : int
        }

    // safe constructor
    let create email visits =
        {emailAddress = email; visits = visits }

    // get data
    let email {emailAddress=e} = e
    let visits {visits=a} = a
```

As before, for each type there is a constructor and a getter for each field.

NOTE: Normally I would define a type outside a module, but because the real constructor needs to be private, I've put the type inside the module and given the module and the type the same name. If this is too awkward, you can rename the module to be different from the type, or use the OCaml convention of calling the main type in a module just "T", so you get `EmailAddress.T` as the type name.

To make a more performant version, we replace `EmailAddress` with a type alias, and `Activity` with a struct, like this:

```
module EmailAddress =

    // aliased type
    type EmailAddress = string

    // safe constructor
    let inline create s :EmailAddress option =
        if System.String.IsNullOrWhiteSpace(s) then
            None
        else if s.Contains("@") then
            Some s
        else
            None

    // get data
    let inline value (e:EmailAddress) :string = e

module ActivityHistory =
    open EmailAddress

    [<Struct>]
    type ActivityHistory(emailAddress : EmailAddress, visits : int) =
        member this.EmailAddress = emailAddress
        member this.Visits = visits

    // safe constructor
    let create email visits =
        ActivityHistory(email,visits)

    // get data
    let email (act:ActivityHistory) = act.EmailAddress
    let visits (act:ActivityHistory) = act.Visits
```

This version reimplements the constructor and a getter for each field. I could have made the field names for `ActivityHistory` be the same in both cases too, but. in the struct case, type inference would not work. By making them different, the user is forced to use the getter functions rather than dotting in.

Both implementations have the same "API", so we can create code that works with both:

```
let rand = new System.Random()

let createCustomerWithRandomActivityHistory() =
    let emailOpt = EmailAddress.create "abc@example.com"
    match emailOpt with
    | Some email  ->
        let visits = rand.Next(0,100)
        ActivityHistory.create email visits
    | None ->
        failwith "should not happen"

let add1ToVisits activity =
    let email = ActivityHistory.email activity
    let visits = ActivityHistory.visits activity
    ActivityHistory.create email (visits+1)

let isCustomerInactive activity =
    let visits = ActivityHistory.visits activity
    visits < 3


// execute creation and iteration for a large number of records
let mapAndFilter noOfRecords =
    Array.init noOfRecords (fun _ -> createCustomerWithRandomActivity() )
    // map it
    |> Array.map add1ToVisits
    // map it again
    |> Array.map add1ToVisits
    // filter it
    |> Array.filter isCustomerInactive
    |> ignore  // we don't actually care!
```

Pros and cons of this approach

One nice thing about this approach is that it is self-correcting -- it forces you to use the "API" properly.

For example, if I started accessing fields directly by dotting into the `ActivityHistory` record, then that code would break when the compiler directive was turned on and the struct implementation was used.

Of course, you could also create a signature file to enforce the API.

On the negative side, we do lose some of the nice syntax such as `{rec with ...}`. But you should really only be using this technique with small records (2-3 fields), so not having `with` is not a big burden.

Timing the two implementations

Rather than using `#time`, this time I wrote a custom timer that runs a function 10 times and prints out the GC and memory used on each run.

```
/// Do countN repetitions of the function f and print the
/// time elapsed, number of GCs and change in total memory
let time countN label f  =

    let stopwatch = System.Diagnostics.Stopwatch()

    // do a full GC at the start but NOT thereafter
    // allow garbage to collect for each iteration
    System.GC.Collect()
    printfn "Started"

    let getGcStats() =
        let gen0 = System.GC.CollectionCount(0)
        let gen1 = System.GC.CollectionCount(1)
        let gen2 = System.GC.CollectionCount(2)
        let mem = System.GC.GetTotalMemory(false)
        gen0,gen1,gen2,mem


    printfn "======================="
    printfn "%s (%s)" label WrappedOrAliased
    printfn "======================="
    for iteration in [1..countN] do
        let gen0,gen1,gen2,mem = getGcStats()
        stopwatch.Restart()
        f()
        stopwatch.Stop()
        let gen0',gen1',gen2',mem' = getGcStats()
        // convert memory used to K
        let changeInMem = (mem'-mem) / 1000L
        printfn "#%2i elapsed:%6ims gen0:%3i gen1:%3i gen2:%3i mem:%6iK" iteration sto
pwatch.ElapsedMilliseconds (gen0'-gen0) (gen1'-gen1) (gen2'-gen2) changeInMem
```

The code sample above is available on GitHub.

Let's now run `mapAndFilter` with a million records in the array:

```
let size = 1000000
let label = sprintf "mapAndFilter: %i records" size
time 10 label (fun () -> mapAndFilter size)
```

The results are shown below:

```
=======================
mapAndFilter: 1000000 records (Wrapped)
=======================
# 1 elapsed:   820ms gen0: 13 gen1:  8 gen2:  1 mem: 72159K
# 2 elapsed:   878ms gen0: 12 gen1:  7 gen2:  0 mem: 71997K
# 3 elapsed:   850ms gen0: 12 gen1:  6 gen2:  0 mem: 72005K
# 4 elapsed:   885ms gen0: 12 gen1:  7 gen2:  0 mem: 72000K
# 5 elapsed:  6690ms gen0: 16 gen1: 10 gen2:  1 mem:-216005K
# 6 elapsed:   714ms gen0: 12 gen1:  7 gen2:  0 mem: 72003K
# 7 elapsed:   668ms gen0: 12 gen1:  7 gen2:  0 mem: 71995K
# 8 elapsed:   670ms gen0: 12 gen1:  7 gen2:  0 mem: 72001K
# 9 elapsed:  6676ms gen0: 16 gen1: 11 gen2:  2 mem:-215998K
#10 elapsed:   712ms gen0: 13 gen1:  7 gen2:  0 mem: 71998K

=======================
mapAndFilter: 1000000 records (Aliased)
=======================
# 1 elapsed:   193ms gen0:  7 gen1:  0 gen2:  0 mem: 25325K
# 2 elapsed:   142ms gen0:  8 gen1:  0 gen2:  0 mem: 23779K
# 3 elapsed:   143ms gen0:  8 gen1:  0 gen2:  0 mem: 23761K
# 4 elapsed:   138ms gen0:  8 gen1:  0 gen2:  0 mem: 23745K
# 5 elapsed:   135ms gen0:  7 gen1:  0 gen2:  0 mem: 25327K
# 6 elapsed:   135ms gen0:  8 gen1:  0 gen2:  0 mem: 23762K
# 7 elapsed:   137ms gen0:  8 gen1:  0 gen2:  0 mem: 23755K
# 8 elapsed:   140ms gen0:  8 gen1:  0 gen2:  0 mem: 23777K
# 9 elapsed:   174ms gen0:  7 gen1:  0 gen2:  0 mem: 25327K
#10 elapsed:   180ms gen0:  8 gen1:  0 gen2:  0 mem: 23762K
```

Now this code no longer consists of only value types, so the profiling is getting muddier now! The `mapAndFilter` function uses `createCustomerWithRandomActivity` which in turn uses `Option`, a reference type, so there will be a large number of reference types being allocated. Just as in real life, it's hard to keep things pure!

Even so, you can see that the wrapped version is slower than the aliased version (approx 800ms vs. 150ms) and creates more garbage on each iteration (approx 72Mb vs 24Mb) and most importantly has two big GC pauses (in the 5th and 9th iterations), while the aliased version never even does a gen1 GC, let alone a gen2.

NOTE: The fact that aliased version is using up memory and yet there are no gen1s makes me a bit suspicious of these figures. I think they might be different if run outside of F# interactive.

What about non-record types?

What if the type we want to optimise is a discriminated union rather than a record?

My suggestion is to turn the DU into a struct with a tag for each case, and fields for all possible data.

For example, let's say that we have DU that classifies an `Activity` into `Active` and `Inactive`, and for the `Active` case we store the email and visits and for the inactive case we only store the email:

```
module Classification =
    open EmailAddress
    open ActivityHistory

    type Classification =
        | Active of EmailAddress * int
        | Inactive of EmailAddress

    // constructor
    let createActive email visits =
        Active (email,visits)
    let createInactive email =
        Inactive email

    // pattern matching
    // not needed
```

To turn this into a struct, I would do something like this:

```
module Classification =
    open EmailAddress
    open ActivityHistory
    open System

    [<Struct>]
    type Classification(isActive : bool, email: EmailAddress, visits: int) =
        member this.IsActive = isActive
        member this.Email = email
        member this.Visits = visits

    // constructor
    let inline createActive email visits =
        Classification(true,email,visits)
    let inline createInactive email =
        Classification(false,email,0)

    // pattern matching
    let inline (|Active|Inactive|) (c:Classification) =
        if c.IsActive then
            Active (c.Email,c.Visits)
        else
            Inactive (c.Email)
```

Note that `Visits` is not used in the `Inactive` case, so is set to a default value.

Now let's create a function that classifies the activity history, creates a `Classification` and then filters and extracts the email only for active customers.

```
open Classification

let createClassifiedCustomer activity =
    let email = ActivityHistory.email activity
    let visits = ActivityHistory.visits activity

    if isCustomerInactive activity then
        Classification.createInactive email
    else
        Classification.createActive email visits

// execute creation and iteration for a large number of records
let extractActiveEmails noOfRecords =
    Array.init noOfRecords (fun _ -> createCustomerWithRandomActivityHistory() )
    // map to a classification
    |> Array.map createClassifiedCustomer
    // extract emails for active customers
    |> Array.choose (function
        | Active (email,visits) -> email |> Some
        | Inactive _ -> None )
    |> ignore
```

The code sample above is available on GitHub.

The results of profiling this function with the two different implementations are shown below:

```
=======================
extractActiveEmails: 1000000 records (Wrapped)
=======================
# 1 elapsed:   664ms gen0: 12 gen1:  6 gen2:  0 mem: 64542K
# 2 elapsed:   584ms gen0: 14 gen1:  7 gen2:  0 mem: 64590K
# 3 elapsed:   589ms gen0: 13 gen1:  7 gen2:  0 mem: 63616K
# 4 elapsed:   573ms gen0: 11 gen1:  5 gen2:  0 mem: 69438K
# 5 elapsed:   640ms gen0: 15 gen1:  7 gen2:  0 mem: 58464K
# 6 elapsed:  4297ms gen0: 13 gen1:  7 gen2:  1 mem:-256192K
# 7 elapsed:   593ms gen0: 14 gen1:  7 gen2:  0 mem: 64623K
# 8 elapsed:   621ms gen0: 13 gen1:  7 gen2:  0 mem: 63689K
# 9 elapsed:   577ms gen0: 11 gen1:  5 gen2:  0 mem: 69415K
#10 elapsed:   609ms gen0: 15 gen1:  7 gen2:  0 mem: 58480K

=======================
extractActiveEmails: 1000000 records (Aliased)
=======================
# 1 elapsed:   254ms gen0: 32 gen1:  1 gen2:  0 mem: 33162K
# 2 elapsed:   221ms gen0: 33 gen1:  0 gen2:  0 mem: 31532K
# 3 elapsed:   196ms gen0: 32 gen1:  0 gen2:  0 mem: 33113K
# 4 elapsed:   185ms gen0: 33 gen1:  0 gen2:  0 mem: 31523K
# 5 elapsed:   187ms gen0: 33 gen1:  0 gen2:  0 mem: 31532K
# 6 elapsed:   186ms gen0: 32 gen1:  0 gen2:  0 mem: 33095K
# 7 elapsed:   191ms gen0: 33 gen1:  0 gen2:  0 mem: 31514K
# 8 elapsed:   200ms gen0: 32 gen1:  0 gen2:  0 mem: 33096K
# 9 elapsed:   189ms gen0: 33 gen1:  0 gen2:  0 mem: 31531K
#10 elapsed:  3732ms gen0: 33 gen1:  1 gen2:  1 mem:-256432K
```

As before, the aliased/struct version is more performant, being faster and generating less garbage (although there was a GC pause at the end, oh dear).

Questions

Isn't this a lot of work, creating two implementations?

Yes! *I don't think you should do this in general.* This is just an experiment on my part.

I suggest that turning records and DUs into structs is a last resort, only done after you have eliminated all other bottlenecks first.

However, there may be a few special cases where speed and memory are critical, and then, perhaps, it might be worth doing something like this.

What are the downsides?

In addition to all the extra work and maintenance, you mean?

Well, because the types are essentially private, we do lose some of the nice syntax available when you have access to the internals of the type, such as `{rec with ...}`, but as I said, you should really only be using this technique with small records anyway.

More importantly, value types like structs are not a silver bullet. They have their own problems.

For example, they can be slower when passed as arguments (because of copy-by-value) and you must be careful not to box them implicitly, otherwise you end up doing allocations and creating garbage. Microsoft has guidelines on using classes vs structs, but see also this commentary on breaking these guidelines and these rules.

What about using shadowing?

Shadowing is used when the client wants to use a different implementation. For example, you can switch from unchecked to checked arithmetic by opening the Checked module. More details here.

But that would not work here -- I don't want each client to decide which version of the type they will use. That would lead to all sorts of incompatibility problems. Also, it's not a per-module decision, it's a decision based on deployment context.

What about more performant collection types?

I am using `array` everywhere as the collection type. If you want other high performing collections, check out FSharpx.Collections or Funq collections.

You've mixed up allocations, mapping, filtering. What about a more fine-grained analysis?

I'm trying to keep some semblage of dignity after I said that micro-benchmarking was bad!

So, yes, I deliberately created a case with mixed usage and measured it as a whole rather than benchmarking each part separately. Your usage scenarios will obviously be different, so I don't think there's any need to go deeper.

Also, I'm doing all my benchmarking in F# interactive. Compiled code with optimizations might have a completely different performance profile.

What other ways are there to increase performance?

Since F# is a .NET language, the performance tips for C# work for F# as well, standard stuff like:

- Make all I/O async. Use streaming IO over random access IO if possible. Batch up your requests.
- Check your algorithms. Anything worse than O(n log(n)) should be looked at.
- Don't do things twice. Cache as needed.
- Keep things in the CPU cache by keeping objects in contiguous memory and avoiding too many deep reference (pointer) chains. Things that help with this are using arrays instead of lists, value types instead of reference types, etc.
- Avoid pressure on the garbage collector by minimizing allocations. Avoid creating long-lived objects that survive gen0 collections.

To be clear, I don't claim to be an expert on .NET performance and garbage collection. In fact, if you see something wrong with this analysis, please let me know!

Here are some sources that helped me:

- The book Writing High-Performance .NET Code by Ben Watson.
- Martin Thompson has a great blog on performance and some excellent videos, such as Top 10 Performance Folklore. (Good summary here.)
- Understanding Latency, a video by Gil Tene.
- Essential Truths Everyone Should Know about Performance in a Large Managed Codebase, a video by Dustin Cambell at Microsoft.
- For F# in particular:
  - Yan Cui has some blog posts on records vs structs and memory layout.
  - Jon Harrop has a number of good articles such as this one but some of it is behind a paywall.
  - Video: High Performance F# in .NET and on the GPU with Jack Pappas. The sound is bad, but the slides and discussion are good!
  - Resources for Math and Statistics on fsharp.org

Summary

> "Keep it clean; keep it simple; aim to be elegant." -- *Martin Thompson*

This was a little experiment to see if I could have my cake and eat it too. Domain modelling using lots of types, but with the ability to get performance when needed in an elegant way.

I think that this is quite a nice solution, but as I said earlier, this optimization (and uglification) should only ever be needed for a small number of heavily used core types that are allocated many millions of times.

Finally, I have not used this approach myself in a large production system (I've never needed to), so I would be interested in getting feedback from people in the trenches on what they do.

The code samples used in this post are available on GitHub.